Université de Montréal

Développement d'une fonction d'évaluation pour le jeu de go

par
Jeffrey Rainy

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Avril, 2005

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Développement d'une fonction d'évaluation pour le jeu de go

présenté par:

Jeffrey Rainy

a été évalué par un jury composé des personnes suivantes:

Geňa Hahn
président-rapporteur

Alain Tapp
directeur de recherche

Stefan Wolf
membre du jury

Mémoire accepté le 30 août 2005

RÉSUMÉ

Le go est un jeu de stratégie à deux joueurs très complexe, pour lequel il n'existe pas de bons adversaires électroniques. Nous présentons une technique pour faire l'apprentissage automatique d'une fonction d'évaluation pour le jeu de go. La technique proposée utilise des réseaux de neurones dont la topologie varie.

Cette fonction, bien qu'insuffisante pour jouer au go par elle-même, est une composante utile dans le développement d'un adversaire électronique. En plus de montrer que l'apprentissage est possible, nous présentons certaines mesures de performance de la fonction apprise. Nous identifions aussi la pertinence relative de certaines caractéristiques de la position.

Finalement, nous décrivons le cadre général dans lequel cette fonction devrait s'intégrer, ainsi qu'une liste partielle des modules nécessaires au développement d'un bon adversaire électronique.

Mots-clés: Intelligence artificielle, Jeu de go, Apprentissage machine, Réseau de neurones.

ABSTRACT

The game of go is a highly complex two-players strategy game. There is currently no computer go programs that can beat the average amateurs. We present a machine learning technique to extract an evaluation function from a set of games played by humans. The proposed method uses neural networks with variable topologies.

This function, although insufficient in itself to play go, is an important part of a computer go program. In addition to showing that learning is possible in this context, we present performance measures for the learned function. We also identify the relative importance of the position characteristics.

Lastly, we describe the general architecture in which such a function can be used, as well as a partial list of necessary modules in the development of a computer go program.

Keywords: Artificial intelligence, Computer go, Machine learning, Neural network.

TABLE DES MATIÈRES

LISTE DES FIGURES

Index

De l'assurance naît l'habileté ;
de l'habileté, l'assurance.

REMERCIEMENTS

Je désire tout d'abord remercier Alain Tapp pour avoir accepté de diriger ma maîtrise. Arrivant ainsi avec ma propre idée de recherche, je risquais fort de me buter à un désintérêt bien justifiable. En revenant aux questions de bases sur le go, sur les réseaux de neurones, et en ayant une perspective plus théorique, beaucoup d'imprécisions et de faussetés ont pu être enlevées du texte. S'il en reste, ce ne sont que les miennes !

J'aimerais aussi remercier toute l'équipe, étudiants et professeurs, du LITQ. D'être dans un lab dont le domaine de recherche n'est pas celui de ma maîtrise m'a donné une toute autre perspective, et des outils bien utiles. Presqu'une deuxième formation ! De plus, ces deux années (presque trois) ont été des plus agréables en leur compagnie.

La patience et le support moral de Nancy doivent aussi être soulignés. Sans ses encouragements, il est bien des occasions où ce travail n'aurait pas avancé autant. Que ce soit d'accepter les horaires irréguliers ou de faire des suggestions pertinentes, sur les démonstrations d'informatique théorique comme sur la lisibilité de ce mémoire, sa présence et ses conseils furent, et sont, grandement appréciés.

CHAPITRE 1

INTRODUCTION

Le go est un jeu de stratégie ayant une très longue histoire. Son origine remonte à plus de 3000 ans. Bien que ses règles soient très simples, il est étonnamment complexe de bien y jouer. C'est sans contredit le plus complexe des jeux de stratégie.

Depuis l'avènement de l'ordinateur, des joueurs informatiques ont été programmés pour la plupart des jeux de stratégie. Pour la très grande majorité des jeux, ces programmes peuvent maintenant battre les meilleurs adversaires humains de chaque discipline. On peut notamment citer les dames et Othello ; à ces jeux, les humains n'ont aucun espoir de gagner, même contre un ordinateur de faible puissance. Le cas des échecs est un cas plus intéressant. Ce n'est que récemment qu'un ordinateur a pu battre le champion du moment. Néanmoins, avec l'amélioration des algorithmes utilisés et avec l'augmentation de la puissance de calcul des ordinateurs, il est fort à parier que les échecs feront bientôt aussi partie des jeux où les ordinateurs battent les humains.

À ce jour, le go est l'exception à cette règle. Malgré une impressionnante recherche dans le domaine, les meilleurs joueurs informatiques programmés jusqu'à maintenant sont encore très faibles. En fait, l'amateur moyen peut battre les meilleurs programmes après une courte période d'apprentissage. Pour ce qui est des professionnels, la possibilité de perdre une partie contre une machine est simplement impensable.

Le développement d'un programme pour jouer au go est donc intéressant car il devrait repousser les limites actuelles de l'intelligence artificielle. Cependant, il ne s'agit pas d'une tâche facile. En effet, la grande différence entre la qualité du go joué par les ordinateurs et celle des grands maîtres est plutôt impressionnante. Elle permet de croire que des années ou même des dizaines d'années de recherche seront nécessaires avant d'obtenir un ordinateur jouant un jeu de niveau professionnel.

Ce mémoire se veut un pas dans cette direction. Une technique pour évaluer globalement une position y est présentée, ainsi que différents résultats obtenus avec cette technique. En aucun cas ce travail ne prétend, à lui seul, donner un algorithme pour bien jouer au go. Cependant, la technique présentée, comme on le verra, a du potentiel et pourra éventuellement faire partie d'un programme jouant au go.

Dans le deuxième chapitre nous présenterons en détails le jeu de go, et parlerons brièvement des programmes existants. Puis, au troisième chapitre, nous couvrirons la base des réseaux de neurones. Nous ne tenterons pas de couvrir l'ensemble du sujet, qui est plutôt vaste, mais tenterons de donner les bases nécessaires à la compréhension de ce projet.

Une fois introduit le jeu de go et les réseaux de neurones, nous présenterons, au chapitre 4, le cœur du projet, l'apprentissage automatique d'une fonction d'évaluation pour le go. Une fonction d'évaluation évalue pour lequel des deux joueurs une position donnée est favorable. Dans notre cas, l'apprentissage automatique se fait à partir d'une très large banque de parties jouées par des humains (373 000 parties).

Pour permettre cet apprentissage, nous avons conçu une technique novatrice pour créer un réseau de neurones dont la topologie dépend de la position sur le goban. Les différentes connections entre les chaînes définissent la structure du réseau. Nous avons opté pour donner de l'information de plus haut niveau à notre réseau. Bien que cela ait demandé un travail substantiel en programmation, la fonction d'évaluation bénéficie de meilleures données, et donne de meilleurs résultats.

Plusieurs autres auteurs ont tenté d'utiliser des réseaux de neurones pour choisir le prochain coups. Notons entre autres, Markus Enzenberger [Enz96], Schraudolph et al. [NNSS94], Cant et al. [RCAD02], ainsi que Donnelly et al. [PDC94]. Cependant, notre utilisation de caractéristiques de plus haut niveau devrait aider le réseau à atteindre de meilleures performances.

L'apprentissage demandant des resources calculatoires plutôt impressionnantes, nous avons distribué la tâche en parallèle, sur seizes machines, pour des durées de deux à six semaines, et ce, à quelques reprises. Incidemment, le temps et les

ressources étant limitées, nous avons dû choisir quelles expériences tenter. Les résultats nous permettent de croire avoir fait un bon choix. Plusieurs optimisations furent aussi nécessaires pour que l'apprentissage se fasse a une vitesse raisonnable. Bien que celles-ci soient décrites sommairement, elles ont demandé une part non-négligeable du travail de développement.

Les résultats sont présentés au chapitre 5. Nous y verrons que l'apprentissage est possible. Pour mesurer la performance, on tente de prédire, à l'aide de notre évaluateur, le bon coup à partir d'une position donnée. Bien que la performance de notre fonction d'évaluation seule soit modeste, en combinant certaines approches, nous obtenons un pouvoir de prédiction assez intéressant.

Finalement, le sixième chapitre donnera une vue plus globale de ce que doit contenir un programme jouant au go, même à un niveau faible. Nous explorons plusieurs types de positions et identifions des éléments stratégiques et tactiques qu'un programme devrait pouvoir reconnaître. Afin de saisir l'utilité de notre technique, nous montrons que certaines de ces techniques bénéficieraient substantiellement d'une bonne fonction d'évaluation.

CHAPITRE 2

LE JEU DE GO

2.1 Qu'est-ce que le go ?

Le go est un jeu de stratégie à deux joueurs. On y joue avec des *pierres* noires et blanches, sur un territoire quadrillé appelé *goban*. Le goban à habituellement 19 lignes sur chaque axe, créant 361 intersections, appelées *points*.

2.2 Les règles du jeu

Cette section présente une version simple des règles, et introduit le vocabulaire utilisé dans ce mémoire. Bien que maintes versions différentes des règles soient utilisées de par le monde, il faut se rappeler qu'à l'exception de cas extrêmement peu fréquents, les différences de règles n'ont pas d'effet. Ces subtiles variations n'affectent généralement que la répétition des positions et le cas où les joueurs ne s'entendent pas sur l'issue de la partie. Nous invitons le lecteur à consulter les règles concises, [Ass91b], ou complètes, [Ass91a], selon son intérêt pour le détail.

2.2.1 La pose des pierres

À tour de rôle, chacun des deux joueurs dépose une pierre de sa propre couleur sur un point du goban ne contenant pas déjà une pierre. Le joueur qui commence la partie jouera les pierres noires, et l'autre les blanches. La figure 2.1 montre le goban, après les six premiers coups d'une partie.

2.2.2 La formation des chaînes

On dit de deux points qu'ils sont adjacents s'ils sont côte à côte sur une même ligne horizontale ou verticale. Deux pierres de la même couleur sur des points adjacents font partie de la même *chaîne*. La figure 2.2 montre deux chaînes blanches,

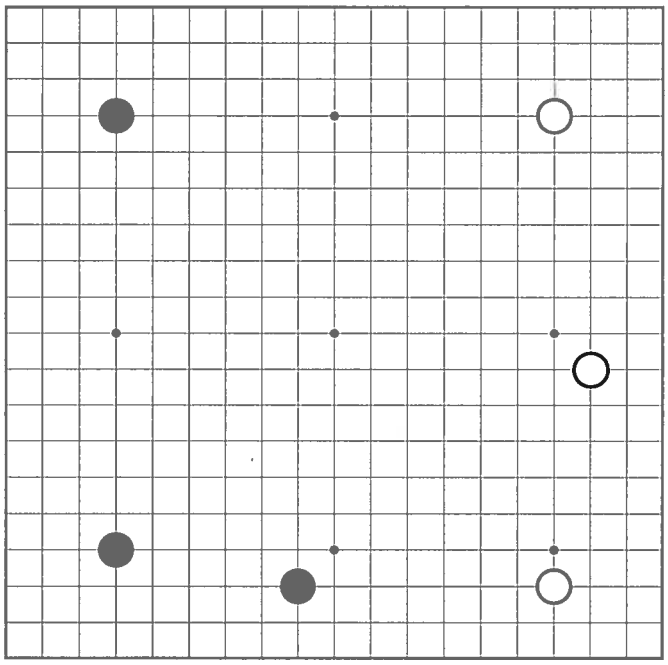

FIG. 2.1 – Une partie après 6 coups

dont l'une est marquée ▲. On voit aussi sur la figure 2.2 trois chaînes noires. On remarquera qu'une chaîne peut être constituée d'une seule pierre, comme c'est le cas de la chaîne noire marqué □.

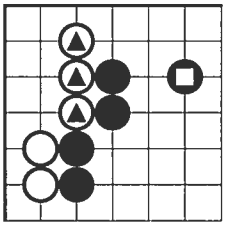

FIG. 2.2 – Un exemple de chaînes

On appelle *liberté* un point libre adjacent à une chaîne. On dit d'une chaîne qu'elle a n libertés, où n est le nombre de libertés adjacentes à une chaîne donnée. On peut voir, sur la figure 2.2 que la chaîne blanche marquée ▲ a 5 libertés, alors

que celle qui n'est pas marquée en a 4. De gauche à droite, les chaînes noires ont respectivement 3, 4 et 4 libertés.

2.2.3 La capture

Après le pose d'une pierre, toute chaîne qui n'a plus de libertés est immédiatement *capturée*. Toutes les pierres de la chaîne capturée sont enlevées du jeu et gardées par le joueur adverse (celui qui vient de poser la pierre) comme *prisonniers*. Il est illégal pour un joueur de jouer de façon à ce que, après son coup, une de ses propres chaînes n'ait plus de liberté. Cependant, si un coup donné effectue une capture, le coup est légal, car la nouvelle chaîne obtient des libertés en capturant la chaîne adverse. La figure 2.3 illustre deux captures possibles.

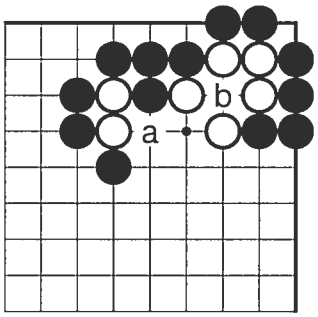

FIG. 2.3 – Deux exemples de captures

On voit sur la figure 2.3 que si le joueur noir joue en a, il capture la chaîne blanche de deux pierres. Aussi, s'il joue en b, il capture la chaîne blanche de trois pierres. C'est pour cette raison qu'il est légal pour le joueur noir de poser sa pierre, en b, là où elle n'aurait pas de liberté sans la capture.

2.2.4 Le ko et la répétition des positions

Il est indésirable pour un jeu de stratégie qu'une partie puisse durer un nombre infini de coups. Pour empêcher la répétition des positions, la règle du *ko* est introduite. Lorsqu'un des deux joueurs effectue la capture d'une seule pierre et que la

pierre qu'il a lui-même posée n'a qu'une seule liberté, il serait possible pour l'adversaire de capturer la pierre posée par le premier joueur. Après cette capture, le jeu se retrouverait dans la même position que deux coups auparavant. Cette situation est appelée ko et est illustré à la figure 2.4.

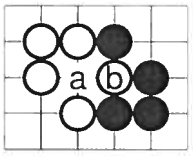

FIG. 2.4 – Un exemple de ko

La règle du ko stipule qu'un joueur ne peut effectuer la capture de la pierre qui vient tout juste d'être posée par son adversaire si cette capture reproduit la position du goban telle qu'elle était au coup précédent. Ainsi, sur la figure 2.4, après le coup du joueur noir en a, le joueur blanc ne peut pas jouer immédiatement en b pour reprendre la pierre noire. Il doit plutôt jouer ailleurs, mais pourra jouer en b plus tard, s'il le souhaite et si son adversaire n'a pas déjà pris ce point.

La règle du *super-ko*, présente dans certaines versions des règles, empêche la répétition des positions, même lorsqu'ells ne sont pas le résultat d'un ko. Il est extrêmement rare qu'il soit nécessaire d'appliquer cette règle. Nous n'en parlerons donc pas plus ici.

2.2.5 Le but du jeu

Le but du jeu consiste à entourer le plus de *territoire* possible. Un territoire est un espace sur le goban où toute pierre de l'ennemi serait nécessairement capturée, éventuellement. Le territoire n'est pas absolu ; c'est la partie du goban que chaque joueur peut défendre contre toute attaque. Il arrive qu'une chaîne soit dans un espace où elle sera nécessairement capturée, malgré toute tentative du joueur pour la sauver. On appelle une telle chaîne un *cadavre*. La figure 2.5 présente un goban avec deux cadavres (chaînes marquées ▲ et □).

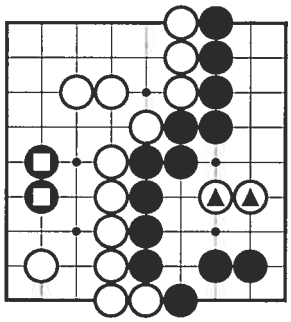

FIG. 2.5 – Un exemple de cadavres

2.2.6 La passe et la fin de la partie

À chaque tour, le joueur peut décider de ne pas jouer, plutôt que de poser une pierre. Ainsi, à la fin de la partie, lorsque chaque joueur croit avoir obtenu tout le territoire qu'il peut obtenir, il passe. Après deux passes consécutives, la partie est terminée si les deux joueurs s'entendent sur les chaînes qui sont des cadavres. Sinon, la partie continue pour permettre aux deux joueurs de démontrer que certaines chaînes de l'adversaire étaient bien des cadavres, en les capturant.

On effectue ensuite le compte des points. Les cadavres de chaque couleur sont ajoutés aux prisonniers de l'autre joueur. Puis, les prisonniers sont remis sur le goban, de façon à réduire le territoire de chaque joueur du nombre de pierres qu'il a perdu. Le joueur avec le plus de points dans son territoire l'emporte. Pour compenser l'avantage de jouer en premier, pour le joueur noir, on donne un certain nombre de points supplémentaires au joueur blanc. On appelle *komi* ce nombre de points. Le komi est habituellement de $6\frac{1}{2}$. Le fait d'utiliser une valeur non-entière pour le komi a pour but d'empêcher une partie nulle.

La figure 2.6 montre une partie qui vient tout juste de terminer. Le joueur noir a 6 prisonniers et le joueur blanc en a 1. On voit que la chaîne ▲ est un cadavre. La figure suivante, 2.7, nous montre le goban après avoir remis les prisonniers dans les territoires, ainsi que le compte de 12 à 17. Si on ajoute au joueur blanc son

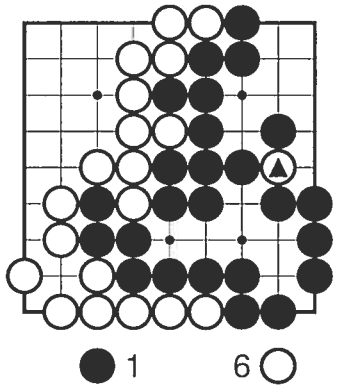

FIG. 2.6 – Une partie terminée

komi de 6 points et demi, il gagne la partie par $1\frac{1}{2}$ points. On remarquera que les joueurs déplacent habituellement les pierres pour faire des territoires rectangulaires et faciliter le décompte. Clairement, cette étape ne change en rien le résultat de la partie.

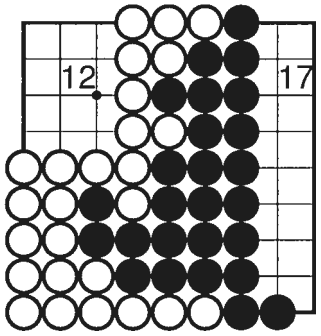

FIG. 2.7 – Le compte final

2.2.7 Le classement et les pierres d'handicap

Le classement des joueurs de go se fait sur une échelle en deux parties, celle des rangs en *kyu* et celle des rangs en *dan*. L'amateur débutant commence entre 20 et

30 kyu, et, en s'améliorant, réduit son rang jusqu'à atteindre 1 kyu. Par la suite, son rang ira de premier dan à septième dan. Les professionnels utilisent une échelle différente, allant de premier dan pro à dixième dan pro. On considère que le plus haut classement amateur, septième dan, correspond environ au plus bas classement professionnel, premier dan pro. La figure 2.8 illustre cette échelle.

FIG. 2.8 – Le classement au go

2.3 Les programmes existants

Plusieurs programmes jouant au go existent déjà. Les plus connus sont *Go4++*, *The Many Faces of Go* [Fot93], *Handtalk*, *Goemate*, *Go Intellect*, *Honte* [Dah99] et *GNU Go*. Leur niveau de jeu tourne habituellement autour de 10 kyu. Cependant, un joueur de 10 kyu qui joue plusieurs parties contre n'importe lequel de ces programmes développe rapidement des techniques efficaces pour les battre. Ce n'est pas tant qu'ils soient faibles, mais bien qu'inévitablement ils feront une, ou plusieurs, grossières erreurs durant la partie.

Bien qu'avec chaque nouvelle version, ces programmes s'améliorent quelque peu, ils sont fondamentalement faibles, et il est fort à parier qu'il faudrait plusieurs nouveaux algorithmes, ou une puissance de calcul beaucoup plus grande, pour les améliorer de façon appréciable.

Il ne faut pas, pour autant, penser que les programmes actuels sont faibles parce que mal programmés, ou parce que peu de recherche a été faite dans le domaine. Ces programmes sont des projets de grande envergure. Pour qu'un programme atteigne ce niveau de jeu, même faible, des années de recherche et développement ont été investies. Notamment, Martin Müller [Mül02] mentionne 5 à 10 année-personnes pour le développement d'un programme compétitif.

CHAPITRE 3

LES RÉSEAUX DE NEURONES

3.1 Qu'est-ce qu'un réseau de neurones ?

Le *réseau de neurones* est un modèle mathématique pour le calcul d'une fonction. Un réseau de neurones est composé d'un ensemble de neurones reliés entre eux par des synapses. Chaque neurone effectue un calcul en fonction de ses entrées, et la sortie d'un neurone est propagée par les synapses vers d'autres neurones. Les synapses, dans le cas qui nous intéresse, ne feront que multiplier leur entrée par une constante. L'ensemble de ces constantes seront appelées *poids* ou *paramètres* du réseau. Aussi, nous nous limiterons aux cas où le réseau ne contient aucun cycle orienté, et où tous les paramètres sont des nombres réels.

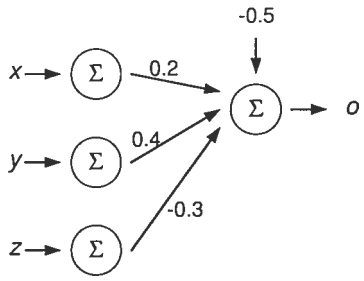

FIG. 3.1 – Un réseau simple

La figure 3.1 montre un exemple de réseau de neurones simple. Le neurone marqué $\sum$ est l'*unité linéaire*. Il calcule la somme de ses entrées, plus une constante w_0. Notre réseau a comme entrées l'ensemble $\{x, y, z\}$, et comme sortie o. Il calcule la fonction $o(x, y, z) = 0.2x + 0.4y - 0.3z - 0.5$. Plusieurs autres types de neurones sont fréquemment utilisés. Ceux-ci seront décrits dans une section ultérieure.

Brièvement, un réseau de neurones peut être vu comme un programme dont le fonctionnement dépend d'un ensemble de paramètres.

3.1.1 Structure usuelle d'un réseau de neurones

La structure habituelle d'un réseau de neurone comprend plusieurs couches successives. Les entrées forment la première couche, suivent plusieurs couchent successives, où les sorties d'une couche sont connectées à toutes les entrées de la couche suivante, et finalement les sorties de la dernière couche sont les sorties désirées du réseau. D'autres topologies sont parfois utilisés selon les besoins. Les explications qui suivent s'appliquent à des réseaux de n'importe quelle topologie, dans la mesure où aucun cycle dirigé n'est formé. Nous discuterons de la topologie utilisée par notre projet dans le prochain chapitre, une fois que les réseaux de neurones auront étés vus.

3.2 Différents neurones

3.2.1 Le perceptron

Le *perceptron* est identique à l'unité linéaire, à la différence près que sa sortie est -1 ou 1 selon que la somme calculée est supérieure ou inférieure à zéro. Ainsi, le perceptron calcule la fonction suivante, où X est l'ensemble des entrées du neurone et w_i sont les paramètres associés :

$$f(X) = \begin{cases} 1, \text{ si } (\sum_X w_i X_i) + w_0 > 0 \\ -1, \text{ si } (\sum_X w_i X_i) + w_0 \leq 0 \end{cases}$$

Un seul neurone perceptron permet de séparer en deux classes un ensemble de points en n dimensions. S'il est possible de faire passer un hyper-plan dans l'espace séparant les points de chacune des deux classes, le perceptron pourra apprendre la classification. Les n poids des synapses et le poids w_0 définissent alors le plan qui sépare les deux classes.

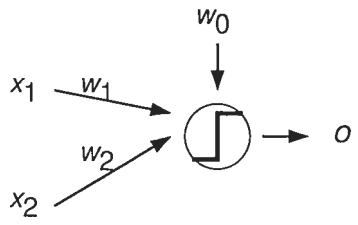

FIG. 3.2 – Le neurone perceptron

3.2.2 Le sigmoïde

Le neurone *sigmoïde* est similaire au perceptron. Comme on le verra plus en détail plus loin, il est souhaitable, pour un neurone, d'avoir une fonction dont la sortie peut être dérivée par rapport à ses entrées. Pour ce faire, on "adoucit" la sortie du neurone perceptron en utilisant une fonction monotone croissante entre -1 et 1. Dans le cadre de ce projet, la fonction $f(x) = \tanh(x)$ a été utilisée. Le choix de la fonction a peu d'importance, dans la mesure où la fonction peut être dérivée, est bornée et monotone croissante. Ainsi, le neurone sigmoïde calcule :

$$f(X) = \tanh((\sum_{X} w_i X_i) + w_0)$$

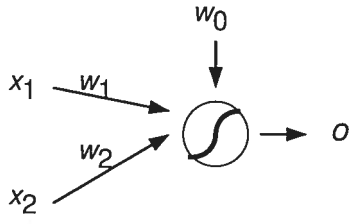

FIG. 3.3 – Le neurone sigmoïde

3.2.3 Le Softmax

Le dernier type de neurone que nous verrons est le neurone *Softmax*. À la différence des autres types vus jusqu'à maintenant, la sortie d'un neurone softmax depend de son entrée mais aussi de l'entrée d'un ou plusieurs autres neurones Softmax. La caractéristique intéressante de ce neurone est que les sorties d'un ensemble de Softmax sont toutes positives et somment à un. Soit $V = \{v_1, \ldots, v_n\}$ un ensemble d'entrées de neurones softmax, les sorties correspondantes de chaque neurone seront $o_i = \frac{e^{v_i}}{\sum_{j=1}^{n} e^{v_j}}$. Pour illustrer, considérons la figure 3.4. Le réseau calculera la fonction suivante :

$$o_i = \frac{e^{w_i x_i}}{\sum_j e^{w_j x_j}}, \text{ car } v_i = w_i x_i$$

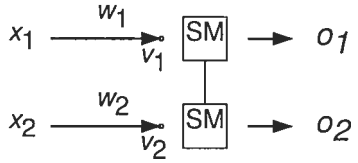

FIG. 3.4 – Réseau simple avec neurone Softmax

3.3 Justifications biologiques

L'utilisation d'une structure telle que le réseau de neurones peut sembler un étrange choix à priori. En fait, la structure du réseau de neurone est basé sur le fonctionnement du cerveau humain. Le cerveau humain contient de l'ordre de 100 milliards de neurones. Ceux-ci émettent un signal plus ou moins puissant, plus ou moins fréquemment. Le taux et la puissance des signaux semblent dépendre des signaux reçu par le neurone, en provenance des autres neurones. Chaque neurone peut être connecté à des milliers d'autres par des synapses.

Ainsi, le réseau de neurone artificiel, appelé communément réseau de neurone, est basé sur le cerveau humain. Il tente d'en reproduire le fonctionnement. Cependant, l'utilisation qui en est faite est celle d'un outil mathématique. Nous l'utilisons parce qu'il permet de trouver une fonction qui minimise certaines caractéristiques, parmi une classe de fonction suffisamment étendue pour certains besoins.

3.4 Objectifs

Un réseau de neurone n'est qu'un outil mathématique. Il calcule une fonction donnée, f', définie par ses paramètres et sa structure. Son utilité réside dans le fait qu'il est possible d'ajuster les paramètres d'un réseau de neurones de façon à ce qu'il minimise une certaine fonction, notamment la différence entre sa sortie, f', et une fonction inconnue, f.

Plaçons-nous dans le contexte où nous voulons apprendre une fonction $f : \mathbb{X} \rightarrow \mathbb{R}$ inconnue, avons un ensemble de paires $(x, f(x))$, et une fonction de coût $c : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$. Nous voulons trouver la fonction f' qui s'approche le plus possible de f, c'est-à-dire la fonction f' telle que sur l'ensemble de toutes les paires, $c(\ f(x), f'(x)\ )$ est minimisée.

Si on définit l'erreur d'apprentissage comme :

$$e = \sum_{x \in X} c(\ f(x), f'(x)\ )$$

on peut trouver un minimum local de e en fonction de f'. Il nous faut d'abord construire un réseau de neurone qui calcule $c(\ f(x), f'(x)\ )$. La figure 3.5 montre ce réseau.

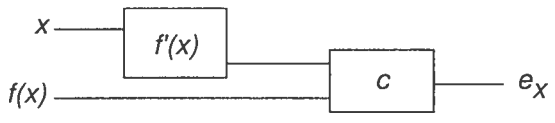

FIG. 3.5 – Le calcul de l'erreur

La technique utilisée par la suite est la descente de gradient ordinaire, que nous présentons dans la prochaine sous-section. Nous définirons aussi la descente de gradient stochastique, une approximation dont le calcul est beaucoup plus rapide.

3.4.1 La descente de gradient ordinaire

La technique de descente de gradient consiste à trouver un minimum local d'une fonction en effectuant les opérations suivantes :

1. Prendre un point du domaine au hasard.

2. Répéter : Calculer le gradient de la fonction au point courant, et déplacer le point dans la direction qui fait baisser le plus rapidement la fonction.

La figure 3.6 illustre cette méthode :

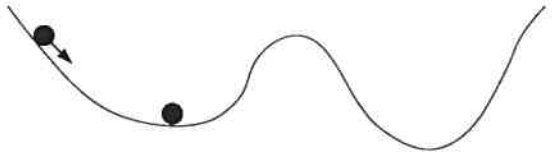

FIG. 3.6 – La descente de gradient ordinaire

La descente de gradient nous fournit donc une méthode générale pour trouver un minimum local, en fonction de certains paramètres, de toute fonction qui peut, de un, être évaluée en tout point du domaine, et de deux, être dérivée par rapport à ces paramètres. Dans la mesure où nous pouvons calculer le gradient de l'erreur d'apprentissage en fonction de chacun des paramètres (les poids) du réseau, nous pourrons trouver un ensemble de poids qui minimisent (minimum local) l'erreur d'apprentissage.

Nous avons rapidement mentionné que la fonction calculée par chaque neurone se devait de pouvoir être dérivée en fonction de son entrée. Le fait de pouvoir calculer la sortie de chaque neurone en fonction de ses entrées nous permettra de calculer la dérivée de la sortie du réseau en fonction de ses entrées. Pour se faire

nous utiliserons la règle de la dérivée en chaîne. Nous illustrerons ce calcul à la figure 3.7.

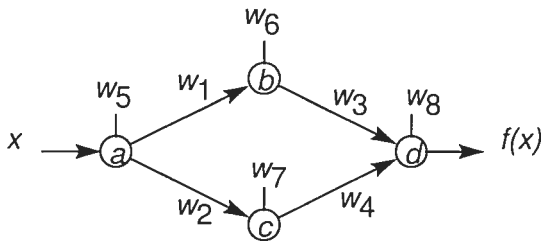

FIG. 3.7 – L'utilisation de la règle de dérivation en chaîne.

On veut obtenir $\frac{\partial f(x)}{\partial w_i}$ pour une valeur donnée de x, pour chaque i, c'est-à-dire la dérivée de la sortie en fonction de chaque paramètre. Nous noterons a_i l'entrée du neurone a, a_o la sortie du neurone a, et $a(a_i, w_5)$ la fonction (dérivable) calculée par le neurone a.

On peut tout d'abord calculer toutes les entrées et sorties des neurones ainsi que $f(x)$, pour le x donné. Puisque le réseau ne contient pas de cycle orienté, cette étape est simple. On propage de gauche à droite les valeurs calculées précédemment.

On peut ensuite, tout aussi facilement, calculer la dérivée de la sortie de chaque neurone en fonction de son entrée et de ses différents paramètres. Ainsi, par exemple, $\frac{\partial d_o}{\partial d_i}$ et $\frac{\partial d_o}{\partial w_8}$ sont facilement obtenues.

Pour obtenir les dérivées de la sortie en fonction des autres paramètres, nous utiliserons la règle de dérivation en chaîne. Ainsi, dans l'exemple qui nous intéresse, on sait que $d_i = w_3 b_o + w_4 c_o$.

Donc, $\frac{\partial d_i}{\partial w_3} = b_0$, et $\frac{\partial d_i}{\partial w_4} = c_0$.

En appliquant $\frac{\partial d_o}{\partial w_3} = \frac{\partial d_o}{\partial d_i}\frac{\partial d_i}{\partial w_3}$ et $\frac{\partial d_o}{\partial w_4} = \frac{\partial d_o}{\partial d_i}\frac{\partial d_i}{\partial w_4}$ on trouve la dérivée de la sortie en fonction de w_3 et w_4. On peut répéter ce processus jusqu'à ce que l'on ait la dérivée de la sortie en fonction de chaque paramètre, ce dont on a besoin pour la descente de gradient.

L'obtention du gradient de la sortie en fonction des paramètres, pour utilisation avec la descente de gradient ou avec une autre technique est habituellement appelée *backpropagation*. De la même façon, on appelle parfois le calcul de la sortie et des dérivées, en fonction des entrées, *propagation*, référant au sens dans lequel le calcul est effectué pour chacune.

Si on peut obtenir le gradient de la sortie du réseau pour chaque échantillon, on peut aussi obtenir le gradient de l'erreur d'apprentissage sur l'ensemble des échantillons en faisant la somme des gradients. Cependant, il est parfois long et coûteux de faire ce calcul pour chaque pas de la descente de gradient. Nous verrons donc une technique plus efficace.

3.4.2 La descente de gradient stochastique

La technique de descente de gradient stochastique est identique à la descente de gradient ordinaire, à la différence que l'on choisit, au hasard, un seul échantillon pour calculer le gradient plutôt que de faire la somme sur tout les échantillons. Sur un grand ensemble d'échantillons, choisis aléatoirement, la descente se fait dans une direction très près de la direction optimale sur l'ensemble des échantillons. Cependant, la descente est beaucoup plus rapide, puisque beaucoup moins de calculs doivent être faits. Dans la pratique, on choisit un pas quelque peu plus petit qu'avec la descente de gradient ordinaire, mais on obtient quand même une excellente hausse de la performance.

Un autre avantage de la descente de gradient stochastique est qu'elle peut, dans certains cas, sortir de minima locaux. En effet, puisque différents échantillons contribuent des gradients différents, il est possible que l'on s'éloigne du minimum local pour un certain nombre d'échantillons, puis que l'on redescende vers un minimum global. La figure 3.8 illustre ce phénomène.

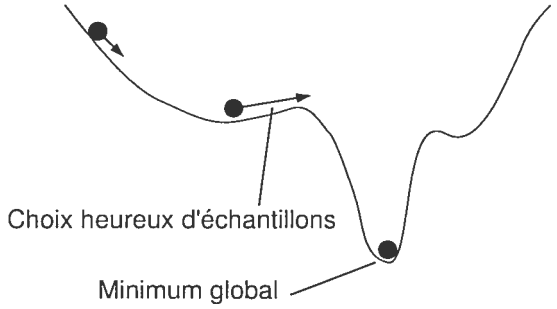

FIG. 3.8 – Sortir de minima locaux avec la descente de gradient stochastique.

3.5 Le problème du surapprentissage

La tâche que nous avons présentée jusqu'à maintenant est la suivante : étant donné une fonction inconnue, f, et un ensemble de points du domaine, X, trouver, parmi une certaine classe, la fonction f' pour laquelle la différence entre f et f' est minimisée sur l'ensemble X.

Nous espérons que la fonction trouvée, f' est proche de f pour des valeurs autres que celles présentes dans X. Cette caractéristique est appelée *généralisation*. Pour le moment, nous n'avons aucune garantie que f s'approche de f' sur ces points. Observons que pour toute fonction f, sur laquelle on mesure un ensemble fini de paires $X = \{(x_i, f(x_i)) \mid 0 < i < k\}$, il est toujours possible de trouver une fonction $f'(x) = a\sin(bx + c)$ telle que $f'(x) = f(x)$ pour $x \in X$. Or, toute fonction ne peut pas être représentée précisément par une sinusoïde.

Le problème du surapprentissage est que f' peut être faite spécifiquement sur mesure pour ressembler à f sur les échantillons présents dans l'ensemble d'apprentissage, X. En simplifiant largement, pour éviter le problème du surapprentissage, on peut limiter la taille et la diversité de la classe de fonctions d'où on tire f', ou s'assurer d'avoir un ensemble d'apprentissage, X, suffisamment grand et varié.

Dans le cas qui nous intéresse, nous ne tiendrons pas compte du surapprentissage, car l'ensemble de données utilisé pour l'entraînement est extrêmement grand

(presque trop), et la classe de fonction plutôt limitée, en comparaison de l'espace de données. Il est donc extrêmement improbable que le problème du surapprentissage ne se présente dans notre cas.

Plus simplement, si notre fonction d'évaluation fonctionne bien sur tous les échantillons, on ne considérera pas qu'elle puisse avoir mémorisé chacun d'eux, car son nombre de paramètres serait insuffisant. Plutôt, nous penserons qu'elle a bien su généraliser.

CHAPITRE 4

UN RÉSEAU DE NEURONES POUR LE GO

Dans ce chapitre nous présentons une architecture particulière de réseau de neurones, pour l'apprentissage d'une fonction d'évaluation des positions. Les techniques utilisées, les approximations choisies, et les optimisations effectuées sont décrites.

4.1 La tâche à accomplir

Le réseau de neurones aura pour tâche de donner une valeur à chaque position selon qu'elle est bonne pour le joueur noir ou pour le joueur blanc. Idéalement, une valeur de 1 serait donnée à toute position d'où le joueur noir peut gagner assurément, et une valeur de -1 serait donnée aux positions d'où le joueur blanc peut gagner. Résoudre ce problème, pour une taille de goban variable, est cependant très complexe, car il est PSPACE-difficile [LS80][1], ou même EXPTIME-complet [Rob83], si les règles japonaises sont utilisées pour le ko.

Dans la pratique, une valeur réelle allant de -1 à 1 peut aussi bien servir, si les positions meilleures ont tendance à avoir des valeurs plus élevées que les positions moins bonnes. Une telle fonction, même approximative, pourrait être très utile pour ordonner les coups dans un arbre de fouille. Nous reviendrons, dans un chapitre futur, sur l'utilité possible de cette fonction. Une autre possibilité aurait été pour le réseau d'apprendre à donner une valeur plus élevée aux positions où noir à un plus grand avantage en terme de points. Nous n'entraînerons pas sur ce point cependant, car nous n'avons aucune donnée sur les territoires effectivement conquis pendant les parties.

[1] Il est trivial de voir que si on pouvait donner une borne polynomiale à la durée d'une partie, le problème serait PSPACE-complet. Une telle borne n'est cependant pas connue.

4.2 La banque de parties

Nous désirons faire l'apprentissage automatique d'un réseau de neurones pour évaluer le résultat d'une partie. Idéalement, nous utiliserions un ensemble de positions pour lesquelles on connaît le gagnant. Cependant, pour la très grande majorité des positions possibles au go, le joueur qui peut gagner est inconnu. Même pour les positions finales où les joueurs passent, rien ne prouve que le gagnant théorique est effectivement celui désigné. En effet, une attaque possible a pu être oubliée par les deux joueurs.

Nous utiliserons plutôt une banque de parties télédéchargée du serveur *No Name Go Server* (`http://nngs.cosmic.org/`)[2]. Cette banque de parties contient 373000 parties de go, jouées en ligne par des joueurs de différents niveaux, remontant jusqu'à juillet 1995. Pour chaque partie, nous connaissons la liste des coups, le handicap au départ, et le classement des joueurs sur ce serveur, lorsqu'ils ont un rang. Cependant, les fichiers ne contiennent pas l'information sur le gagnant ni le score final. En fait, rien ne garantit même que la partie n'ait été terminée. Nous utiliserons donc la seule information disponible, le coup choisi par le joueur, en faisant l'hypothèse qu'il s'agit du meilleur coup.

Clairement, cette hypothèse s'avère fausse souvent. En effet, tout joueur commet fréquemment des erreurs. Néanmoins, puisque le coup joué est souvent parmi les meilleurs coups, et puisque nous tentons d'apprendre une fonction d'évaluation qui estime la valeur d'une position, nous utiliserons tout de même cette hypothèse. Nous verrons plus loin qu'il suffit que ce coup soit meilleur qu'un coup aléatoire pour qu'il nous fournisse une certaine information.

---

[2]Ces parties sont disponibles publiquement sur le site web, sans mention de Copyright. Plusieurs courriels demandant l'autorisation d'utiliser cette banque de parties sont restés sans réponse. Pour cette raison, nous ne fournissons pas ces parties.

4.3 Les entrées du réseau d'évaluation

On veut un réseau qui évalue le résultat probable de la partie à partir d'une position sur le goban. Plutôt que de prendre, en entrée, une matrice de valeurs {blanc, noir, vide} pour chaque point, nous avons tenté d'utiliser des données de plus haut niveaux, c'est-à-dire des données qui représente les concepts importants pour un joueur de go.

En partant d'une position, dans un premier pré-traitement, plusieurs de ces données sont extraites pour l'évaluation. En voici la liste :

- La couleur de chaque chaîne

  La chaîne appartient à un joueur où à l'autre. Clairement nécessaire pour évaluer le score.
- La taille de chaque chaîne

  De la même façon, l'état d'une chaîne plus grande devrait avoir un impact plus grand sur le score.
- Le nombre de liberté de chaque chaîne

  Une chaîne ayant plus de libertés est plus difficile à capturer. C'est une information importante, car l'état vivant ou capturée de la chaîne en dépend souvent directement.
- Les connections entre les chaînes

  Les différentes connections que l'on peut avoir entre deux chaînes constituent un élément très important au go. Par connection, on entend que les deux chaînes ont des pierres placées l'une par rapport à l'autre d'une façon telle que la fusion des deux chaînes, en jouant un coup adjacent aux deux, est relativement facile. La figure 4.1 montre les différentes connections. Bien que le score soit moins directement affecté par le lien entre les chaînes de couleurs différentes, ce lien peut affecter directement la survie des chaînes. Aussi, nous avons aussi trouvé les liens entre chaînes blanches et noires, même s'ils peuvent difficilement être appelés connections. La figure 4.2 montre les "connections" entre chaînes blanches et noires. Dans ces deux figures, on no-

tera un ensemble de petits ronds noirs marquant certains points. Ces ronds indiquent les points qui doivent être libres pour que la connection soit présente. En effet, il n'y aurait aucun sens de considérer connectées deux chaînes noires séparées par une chaîne blanche.

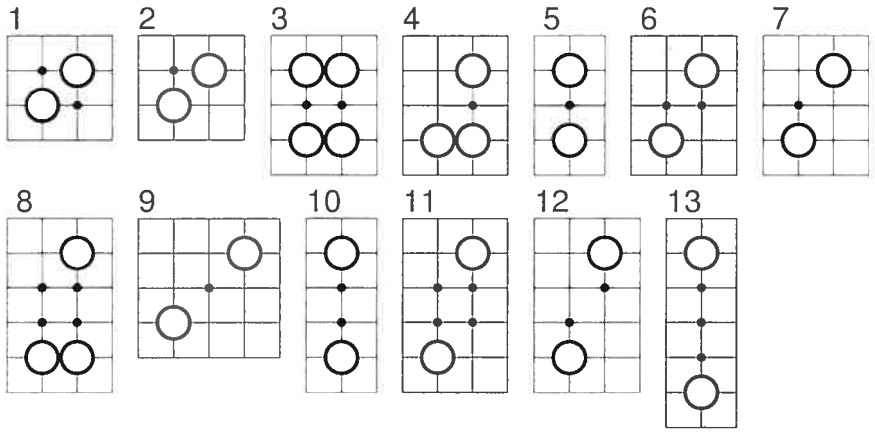

FIG. 4.1 – Les connections entre chaînes de même couleur.

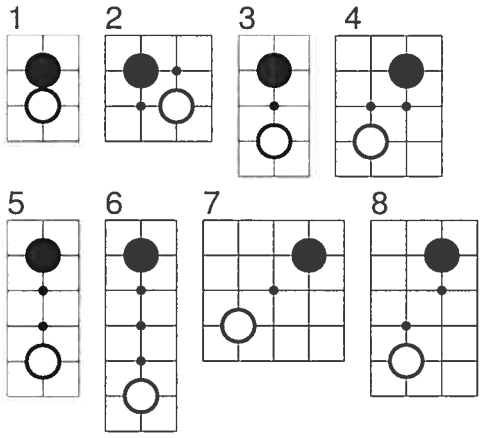

FIG. 4.2 – Les connections entre chaînes de différentes couleurs.

On espère que l'ensemble des connections ainsi que les caractéristiques des chaînes donnent une image fidèle du goban et permettent une évaluation du score. Pour arriver à ce résultat, il nous manque clairement une information importante : la position des chaînes par rapport aux bords du goban. Pour

contrer cette lacune, nous avons aussi trouvé les connections entre les chaînes et les bords du goban. La figure 4.3 montre les connections considérées. Nous avons considéré les chaînes à plus de 4 points du bord comme non connectées avec la bordure, car la très grande majorité des batailles pour le bord du goban se produisent à l'intérieur des 4 premières lignes.

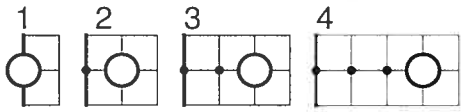

FIG. 4.3 – Les connections entre les chaînes et le bord du goban.

Revenons maintenant à l'information extraite du goban.

- Le nombre maximal de libertés qui peuvent être gagnées en un seul coup
  Une chaîne qui peut obtenir plusieurs libertés supplémentaires, en un seul coup, peut être plus difficile à capturer. Nous avons donc ajoutée cette information.
- L'influence
  L'influence désigne généralement l'effet global d'une chaîne sur les points libres du goban. Pour pouvoir évaluer le score, nous avons choisi une définition plus précise. Nous définissons l'influence d'une chaîne comme le nombre de points sur le goban dont la *distance de Manhattan*[3] à la chaîne est inférieure à la distance à tout autre chaîne. Ainsi on considère que chaque chaîne affecte tous les points situés plus près d'elle que de toute autre chaîne. La figure 4.4 montre l'influence de la chaîne marquée $\triangle$.
- L'information sur les yeux
  Pour pouvoir ne pas être capturée par l'adversaire, une chaîne sur le goban doit, lorsque la partie termine, entourer au moins deux régions vides distinctes. On appelle ces régions des *yeux*. Avant que la partie ne termine, il est difficile de savoir quelles chaînes auront deux yeux. Cependant, on peut

[3]La distance de Manhattan est la somme de la distance séparant deux points, à l'horizontale et à la verticale. Dans notre cas, le nombre de point adjacents séparant deux chaînes.

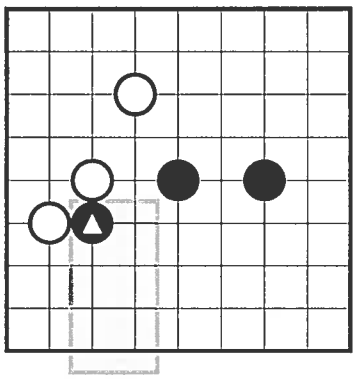

FIG. 4.4 – L'influence d'une chaîne.

parfois identifier un œil, ou, à tout le moins, certaines structures qui pourraient devenir un œil. Nous avons implémenté l'algorithme de Benson [Ben76], et utilisons plusieurs des données servant à identifier les yeux.

Il est plus facile de travailler avec des réseaux où les valeurs sont bornées entre -1 et 1. Ainsi, les entrées ont été converties avec des fonctions monotones croissantes, pour les ramener entre 0 et 1. La figure 4.5 montre les fonctions utilisées pour la taille, l'influence et le nombre de libertés. Nous avons choisi les fonctions de façon à ce qu'en moyenne un nombre de libertés, une influence et une taille usuels donnent tous une valeur de 0.5 (marqués ♦ dans la figure). De plus, nous avons utilisé des fonctions non-linéaires pour donner plus d'impact aux variations de petites valeurs, car les petites valeurs de libertés, d'influence ou de taille peuvent avoir un impact majeur sur la vie d'une chaîne.

Dans la prochaine section, nous verrons comment ces multiples données d'entrée sont utilisées.

4.4 L'architecture utilisée

Puisque le nombre de chaînes sur le goban varie et que l'information pertinente est associée aux chaînes, le nombre d'entrées variera. Notre choix d'entrées nous empêche donc d'utiliser une topologie fixe pour le réseau. De plus, les connections sont en nombre variable et constituent un élément important de l'évaluation.

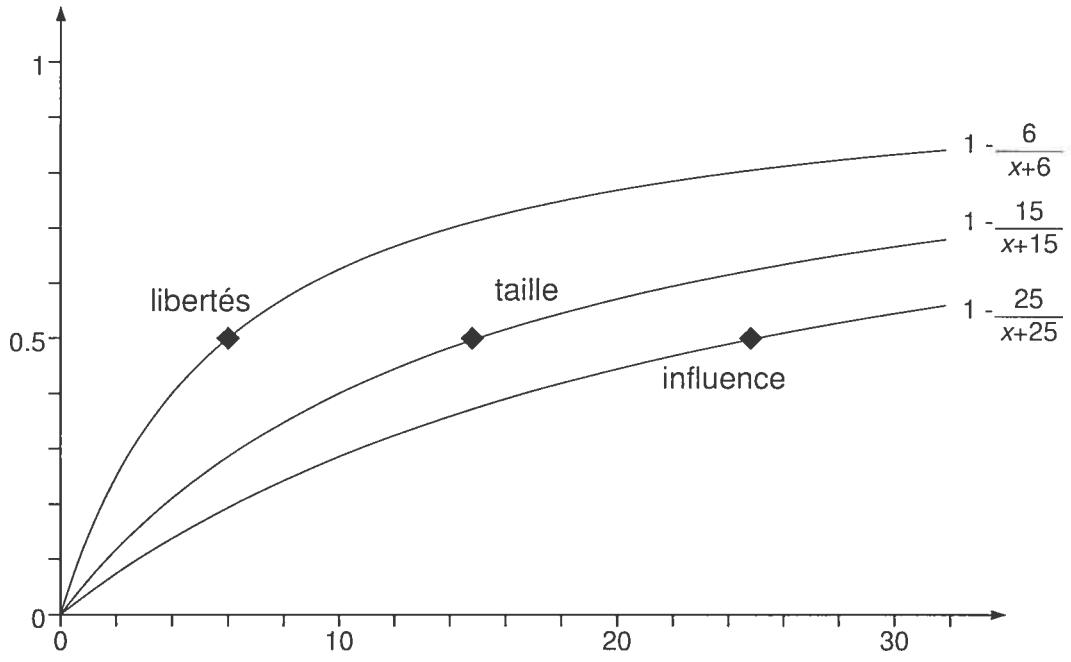

FIG. 4.5 – La conversion des valeurs d'entrée.

Nous avons donc choisi de donner une forme différente à notre réseau pour chaque goban évalué. La forme dépendra de l'ensemble des connections, et les poids seront pris dans une banque de poids de taille fixe. Lors de l'apprentissage, tout changement à la valeur d'un poids sera reflété dans la banque de poids. Nous expliquerons plus en détails dans les sous-sections qui suivent cette procédure, et montrerons que la descente de gradient fonctionne bien avec une telle organisation.

En entrée, nous aurons un bloc de valeurs par chaîne. Chaque bloc d'entrées aura chacune des valeurs présentées précédemment, pour sa chaîne. De plus, un bloc d'entré sera ajouté pour chacun des quatres cotés du goban. De la même façon que deux chaînes seront connectées, on pourra ainsi connecter une chaîne et un bord du goban. Les valeurs d'entrée pour les blocs correspondant aux bords du goban seront des paramètres qui pourront être appris, comme les poids.

4.4.1 Le graphe de connection

Le graphe de connection est un graphe où les sommets sont les chaînes et où une arête est présente entre le sommet a et b si la chaîne a est connectée à b par un des motifs montrés plus haut. Dans ce cas, l'arête est étiqueté par la connection correspondante. Ce graphe décidera de la topologie du réseau de neurones. Lorsque deux connections relient deux chaînes, la plus solide est conservée. Une connection est plus solide qu'une autre, s'il est plus facile pour le joueur de faire se rejoindre les deux chaînes, lorsque l'adversaire tente de les séparer. Nous avons présenté les connections dans l'ordre de la plus solide à la moins solide. La figure 4.6 montre un exemple de goban avec les connections identifiées.

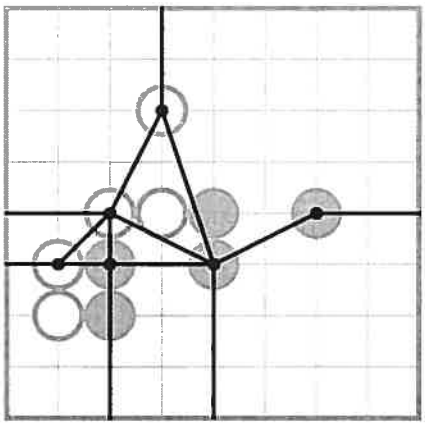

FIG. 4.6 – Un graphe de connection.

4.4.2 Le bloc de neurones

Notre réseau aura un nombre fixe d'*étages*. Par étage, nous entendons un ensemble de couches de neurones successives logiquement reliées entre elles. Les entrées du réseau forment les entrées du premier étage. Chaque bloc de neurones, que nous définirons sous peu, occupe un étage de profondeur, mais plusieurs blocs se partagent le même étage.

Si deux chaînes sont connectées sur le goban, un bloc de neurones les reliera dans le réseau de neurones. Ce bloc prendra en entrée les données des deux chaînes et aura en sortie autant de valeurs que le nombre d'entrées pour une chaîne. De cette façon, on peut propager la sortie du bloc aux deux chaînes correspondantes, mais à l'étage suivant.

Chaque bloc a un nombre fixe de niveaux, et un nombre fixe de neurones sur chaque niveaux. Les poids de chaque synapse du bloc sont spécifique à chaque bloc. Ainsi, si le même bloc est utilisé entre deux paires de chaînes différentes, le même poids est utilisé deux fois. Lors de l'apprentissage, toute modification à un poids affecte le poids du bloc. La figure 4.7 illustre un bloc de connection. On voit que les blocs utilisent des paramètres notés $\omega_{i,j}$ où i est le numéro de la connection entre les deux chaînes et j l'indice du poids dans le bloc. La figure 4.8 montre l'architecture du réseau, c'est-à-dire un ensemble d'entrées, un bloc et la sortie.

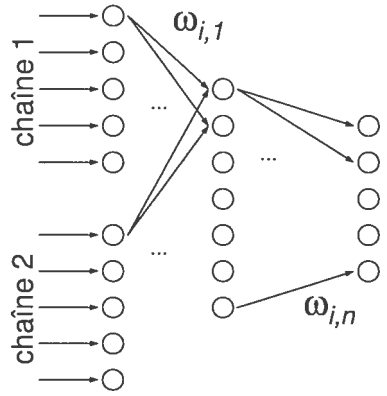

FIG. 4.7 – Un bloc de connection.

4.4.3 La sortie

Nous avons décrit un ensemble d'entrées et plusieurs blocs connectant les entrées des différentes chaînes entre elles. Pour obtenir la sortie désirée donnant une estimation du gagnant, il faut combiner les différentes valeurs à la sortie des blocs de

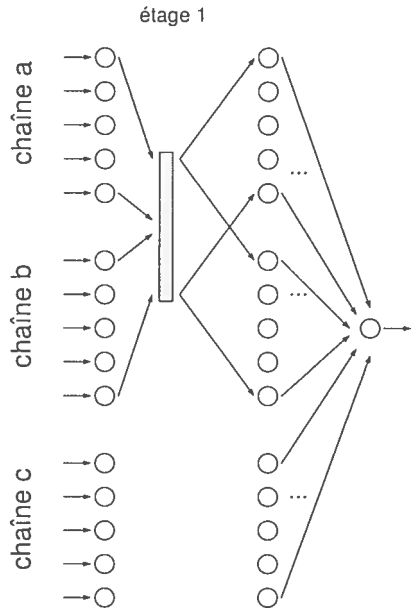

FIG. 4.8 – L'architecture du réseau.

connection. Nous avons opté pour une combinaison linéaire des différentes valeurs. Puisque tous les blocs ont une importance égale, la pondération ne se fera que sur les sorties de chaque bloc. Ainsi, un nombre de poids égal au nombre de sorties servira à sommer tous les blocs vers un seul neurone de sortie.

4.4.4 L'utilisation de plusieurs étages

L'idée derrière le choix d'architecture est que l'on peut estimer les points qu'une chaîne contribuera au résultat final en regardant les caractéristiques de la chaîne et des différentes chaînes qui l'entourent. De la même façon, il semble logique de penser qu'en regardant aussi les chaînes qui entourent ces chaînes, on puisse arriver à une meilleure estimation. Avec l'architecture proposée, regarder une chaîne plus loin correspond à ajouter un étage supplémentaire de blocs, de la même façon qu'on a construit le premier. Nous avons donc aussi entraîné le réseau avec plusieurs étages. La figure 4.9 illustre cet arrangement.

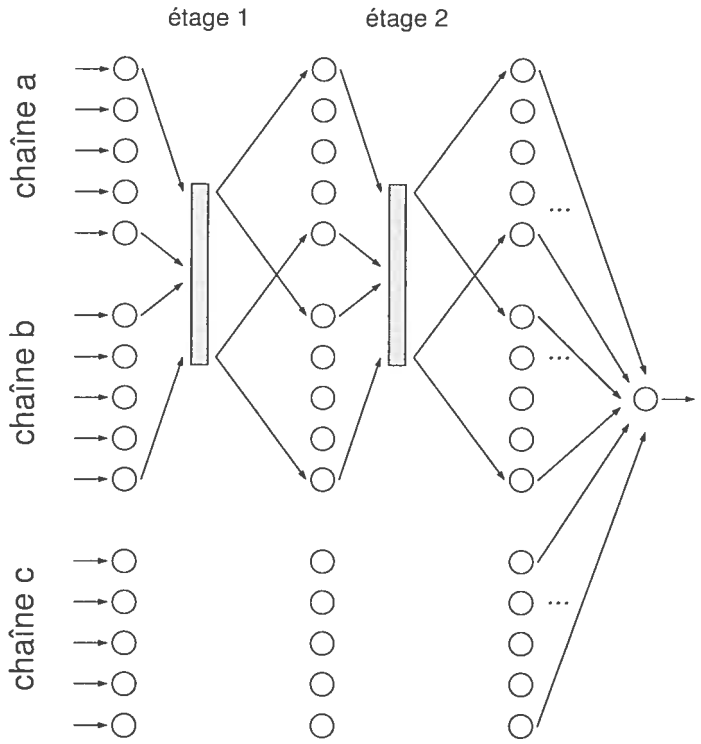

FIG. 4.9 – L'architecture à plusieurs étages.

Deux possibilités apparaissent pour les paramètres du deuxième étage. Une première possibilité est de faire l'apprentissage en utilisant les mêmes blocs de paramètres que pour le premier étage. Dans ce cas, on force le réseau à apprendre un ensemble de poids qui permet l'évaluation sur deux étages identiques. Cela correspond à dire au réseau qu'il existe une façon d'évaluer le score en appliquant le même calcul aux chaînes proches et aux chaînes un peu plus éloignées. Bien que cela réduise le nombre de paramètres, il semble y avoir une meilleure solution. Une deuxième possibilité est de faire l'apprentissage avec un second ensemble de poids différents pour le deuxième étage. Le désavantage majeur est que le nombre de poids double et rend l'apprentissage plus long.

Il existe cependant une solution à ce dernier problème. Après l'apprentissage à

un étage, l'information qui est sommée pour obtenir une évaluation du goban est pertinente. Elle peut donc servir de base à un apprentissage successif. Ainsi, on peut faire l'apprentissage sur un étage, puis ajouter un deuxième étage et laisser le réseau apprendre les paramètres de ce deuxième étage, tout en modifiant ceux du premier. La question qui se pose alors est de savoir comment initialiser les valeurs du deuxième étage lors de sa création.

Une possibilité serait de l'initialiser avec des poids aléatoires de faibles valeurs (par exemple entre -0.1 et 0.1). Cela permet l'apprentissage. Cependant, pendant que ces poids convergent vers un minimum local, les poids du premier étage ont amplement le temps de s'éloigner de leur valeur initiale qui contenait possiblement une information utile. Une meilleure solution consisterait à utiliser un ensemble de blocs identité pour le deuxième étage. Un bloc identité serait un bloc dont les paramètres font que l'ajout d'un étage de ces blocs ne change pas l'évaluation faite par le réseau. Ainsi, après l'ajout du bloc, l'évaluation serait aussi bonne qu'auparavant, mais aurait un ensemble supplémentaire de paramètres libres pour encore s'améliorer.

Malheureusement, avec l'architecture utilisée, le bloc identité n'existe pas. En effet, deux chaînes ayant des caractéristiques différentes auront, en sortie du bloc, des caractéristiques identiques. Cette découverte souligne certaines possibilités qui pourraient encore être explorées, notamment, de modifier l'architecture pour que le bloc identité existe. Que notre classe de fonctions possibles pour les blocs ne contienne pas la fonction identité indique peut-être que la classe est trop restrictive. De la même façon, l'architecture présentée a une faiblesse qui pourrait grandement être révisée : Soit deux chaînes a et b. La valeur produite par le bloc sur ces deux chaînes est différente selon l'ordre des deux chaînes en entrée du bloc. Or, le réseau est construit en plaçant les chaînes dans l'ordre de création sur le goban. Comme cet ordre est modifié de façon arbitraire lorsque deux chaînes sont connectées, on peut considérer que les chaînes sont placées de façon arbitraire en entrée. Ainsi, le même goban aura plusieurs évaluations différentes possibles selon l'ordre des chaînes. Ces deux faiblesses limitent la qualité de l'évaluation, mais n'empêchent

tout de même pas l'apprentissage, comme nous le verrons plus loin.

4.4.5 La descente de gradient dans l'espace des paramètres

Les descentes de gradient ordinaire et stochastique sont habituellement utilisées avec des réseaux de neurones dont la structure est fixe. On pourrait se demander si ces deux techniques nous mèneront bien à un minimum local dans le cas où la structure du réseau est différente pour chaque entrée.

Pour se convaincre que la descente de gradient nous mène bien à un minimum local, il suffit de regarder l'ensemble formé par l'algorithme pour trouver les données des chaînes, bâtir le réseau, et évaluer l'estimation du score. L'algorithme et le résultat calculé peuvent être vus comme une fonction. Cette fonction à un certain nombre de paramètres libres : les poids associés à chaque synapse dans chaque bloc. Elle provient donc d'une classe de fonctions. Puisque nous pouvons dériver cette fonction selon chacun de ses paramètres, la descente de gradient nous garantit de trouver un minimum local de l'erreur dans cette classe.

4.5 L'entraînement

Une caractéristique principale du projet est que nous avons un très large ensemble de coups joués, mais rien d'autre. Nous n'avons pas de données sur la valeur de chaque position, que notre fonction tentera d'estimer. Nous savons seulement quel coup le joueur a choisi. Implicitement, nous avons donc de l'information sur la valeur des positions et il nous faut trouver une façon efficace de faire l'apprentissage.

Une possibilité considérée fut la suivante : appelons le coup joué par l'humain "coup joué". Prendre, pour chaque autre coup possible, la position résultante après ce coup. Évaluer la valeur de chaque position selon l'estimateur actuel, et garder le coup donnant le meilleur score : "coup choisi". On voudrait que ce coup soit le coup joué. En se rappelant l'hypothèse imparfaite selon laquelle le coup joué est le meilleur, notre estimateur se trompe s'il choisi un coup différent. Son erreur est alors d'autant plus grande que la différence de valeur entre le coup choisi et le coup

joué. On peut donc entraîner en minimisant la différence entre la valeur du coup choisi et du coup joué.

Cette technique a comme désavantage qu'il faut estimer la valeur d'un grand nombre de positions puisque, en moyenne, autour de 200 coups sont possibles pour chaque position. Il nous en coûte 200 étapes de propagation et de rétro-propagation, pour chaque pas d'apprentissage.

Une technique plus rapide peut être utilisée. Prenons le coups joué, évaluons la position après ce coup. Prenons un coup aléatoire, et évaluons à nouveau la position. Un bon estimateur devrait donner une valeur supérieure à la position joué qu'à celle choisie aléatoirement. On peut donc entraîner en minimisant la différence entre le coup choisi et le coup joué.

Cette technique prend uniquement 2 passes de propagation et de rétro-propagation. Nous pourrons donc faire 100 fois plus de pas d'apprentissage. Cependant, les pas seront moins précis, car on entraîne uniquement pour jouer un coup meilleur que le coup aléatoire. Chaque pas devrait néanmoins aller dans une direction qui minimise l'erreur, car selon les hypothèses posées, un coup aléatoire est toujours moins bon que le coup joué.

4.5.1 La fonction de coût

Tel que décrit dans le chapitre précédent, on souhaite minimiser le coût associé à la valeur d'évaluation du réseau pour une entrée donnée. Dans notre cas cependant, nous n'avons pas l'évaluation désirée. Nous calculerons donc le coût à partir de l'évaluation pour le coup joué et le coup aléatoire ou choisi, selon le cas. Plutôt que de minimiser la différence, nous avons plutôt utilisé un neurone Softmax, et maximiserons la valeur associé au goban dont l'évaluation doit être supérieure (coup joué). La figure 4.10 illustre l'utilisation du neurone Softmax.

Dans les faits, nous ne souhaitons pas que les valeurs d'évaluation puissent être très grandes, car cela cause des problèmes de précision numérique. En autant que les valeurs peuvent rejoindre une plage suffisamment large de valeurs, elles pourront efficacement être comparées l'une par rapport à l'autre. Aussi nous avons fait tendre

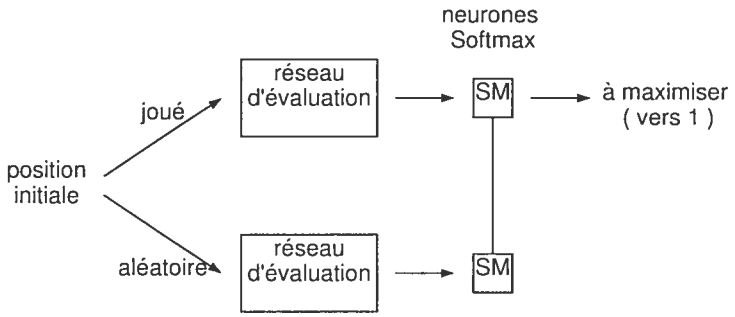

FIG. 4.10 – L'utilisation du neurone Softmax.

la sortie du neurone Softmax vers 0.9, ou 0.95 selon l'expérience, plutôt que vers 1. De cette façon, les valeurs d'évaluation ne tendaient pas vers l'infini.

4.5.2 Le parallélisme

Chaque pas d'apprentissage demande une bonne quantité de calcul. En effet, il faut préparer la position, jouer le coup joué et un coup aléatoire, évaluer les caractéristiques des chaînes pour chaque position, faire la propagation, la rétro-propagation et finalement modifier les poids dans les banques de poids. De plus, pour que l'apprentissage par descente de gradient stochastique soit le plus efficace possible, il faut idéalement choisir les positions de façon aléatoire parmi la banque de parties. De se rendre au *n*ième coup d'une partie demande aussi une quantité non négligeable de calcul.

Notons tout d'abord qu'il est inacceptable de faire l'apprentissage sur des coups séquentiels de la même partie. En effet, pour que la descente de gradient stochastique se comporte comme la descente de gradient ordinaire, il faut lui donner des échantillons qui sont uniformément distribué sur l'ensemble des parties. Les positions successives de la même parties ont des caractéristique communes, et le réseau n'apprendrait que celles-ci. De devoir utiliser des positions variées affecte la méthode de parallélisation.

Pour régler se problème, on peut préparer en mémoire un grand nombre (1000) de positions, provenant de parties différentes, puis faire l'apprentissage sur un coup de chacune, de façon séquentielle. L'avantage est que le goban correspondant à chaque position est alors dans la même partie, mais un coup plus tard, prêt pour le prochain apprentissage. Ainsi, le calcul des caractéristiques se fait de façon itérative sur un goban où un seul coup supplémentaire vient d'être joué. Si on s'arrange pour que les 1000 parties ne soient pas toute dans la même phase (début de partie, fin de partie) au même moment, l'ensemble des coups appris est reparti presque uniformément sur l'ensemble des coups. Cette optimisation à elle seule enlève la majorité du temps de calcul associé à préparer les gobans.

Notons que cette technique à demandé un temps assez important pour son développement. En effet, chaque caractéristique utilisée sur le goban a dû être calculée de façon itérative. Nous avons tenté de ne pas recalculer tout ce que l'on peut se rappeler du goban précédent, lorsqu'une seule pierre est ajoutée.

Une autre optimisation utilisée consiste à se servir de plusieurs machines pour faire la descente de gradient stochastique. La façon la plus évidente aurait bien sûr été de soumettre à chacune de n machines une fraction des parties. Par la suite, après chaque pas d'apprentissage, les modifications aux poids sont propagées à toutes les machines. Cette façon de faire aurait deux désavantages : la communication nécessaire est passablement élevée et le temps de faire un pas varie largement selon la complexité de la position. Ainsi les machines auraient souvent attendu que la communication soit complète ou que l'évaluation d'une position plus complexe soit terminée.

Nous avons plutôt choisi de laisser chaque machine faire la descente de gradient de façon indépendante pour un certain temps. Après cette durée, toutes les machines échangent leur ensemble de poids, la moyenne est calculée, et chacune repart de ce nouveau point dans l'espace des paramètres. Dans la mesure ou le temps entre les échanges est suffisamment petit, la descente de gradient en est accélérée. En fait, on retrouve un peu de la précision de la descente de gradient ordinaire puisque plusieurs échantillons contribuent à chaque pas.

Dans le prochain chapitre, nous présenterons les résultats obtenus avec cette architecture.

CHAPITRE 5

LES RÉSULTATS

Dans ce chapitre, nous présenterons les différents résultats obtenus avec deux façons différentes d'entraîner le réseau. La performance de la fonction d'évaluation est analysée et comparée avec une méthode simple, avec et sans arbre de fouille. Chacun de ces entraînements représente plusieurs semaines de temps de calcul.

5.1 Le pouvoir prédictif

La mesure d'efficacité utilisée est le *pouvoir prédictif*. Pour le définir, observons une position donnée, prise dans une partie jouée par deux joueurs humains. Considérons que le coup joué par l'humain est le meilleur coup possible. On voudrait que l'évaluation donnée à la position résultante après ce coup soit plus grande que celle donnée à chacune des autres. L'indice de prédiction, pour une position donnée, est le pourcentage des coups dont l'évaluation est supérieure à celle du coup choisi par le joueur humain.

Le pouvoir prédictif sera défini comme la moyenne de l'indice de prédiction, sur un large ensemble de positions variées. Un meilleur pouvoir prédictif correspond à une valeur inférieure. Un estimateur parfait aurait une valeur très proche de zéro, lorsque évalué sur une partie jouée par des joueurs jouant très bien.

Nous utiliserons le pouvoir prédictif comme mesure de la performance, même si les coups que notre estimateur tentera de prévoir n'ont pas tous été joués par des joueurs parfaits, ou mêmes très bons. Cette évaluation de la performance perdrait son sens si notre estimateur devenait assez bon pour souvent prédire un coup meilleur que celui joué. Malheureusement, ce n'est pas encore le cas, et le pouvoir prédictif reste une bonne mesure de la performance.

5.2 Résultats avec petit ensemble d'entrées

Les expériences ont été faites avec 16 machines faisant la descente de gradient stochastique décrite au chapitre précédent. Le nombre d'étages, le nombre de neurones par étage des blocs et les entrées ont varié.

5.2.1 Petit ensemble d'entrées sans Minimax

Le premier résultat est obtenu avec un réseau à deux étages, et un ensemble limité d'entrées, c'est-à-dire : la couleur, le nombre de libertés, la taille, l'influence, l'incrément de libertés et les connections. Le bloc de connection utilisé a 3 niveaux ayant respectivement 10, 7 et 5 neurones. Le nombre d'entrées, 10, et de sorties, 5, découle directement du nombres de caractéristiques choisies comme entrées.

L'apprentissage a été fait à un étage, puis un second étage initialisé aléatoirement a été ajouté lorsque le pouvoir prédictif ne semblait plus s'améliorer. L'apprentissage a continué à deux étages jusqu'à ce que l'amélioration cesse à nouveau. Il est à noter que, lors de l'ajout du deuxième étage, une certaine baisse de pouvoir prédictif est observée, puisque le premier étage est alors entraîné pour évaluer seul. Cependant, le réseau à deux étages dépasse rapidement la performance de celui à un étage.

Le pouvoir prédictif moyen obtenu est de 0.198. Cette mesure a été effectuée sur vingt parties choisies aléatoirement dans la banque de partie.

Il est intéressant d'observer si la performance de la fonction d'évaluation est constante sur toute la partie et si elle est la même pour le joueur noir et pour le joueur blanc. La figure 5.1 montre le pouvoir prédictif en fonction de ces deux paramètres.

5.2.2 Petit ensemble d'entrées avec fouille Minimax

Les plus fréquentes utilisations d'une fonction d'évaluation, pour une position dans un jeu, impliquent une fouille dans un arbre, pour trouver un bon coup. La question évidente qui se pose alors est : "Peut-on, en utilisant une fouille, obtenir

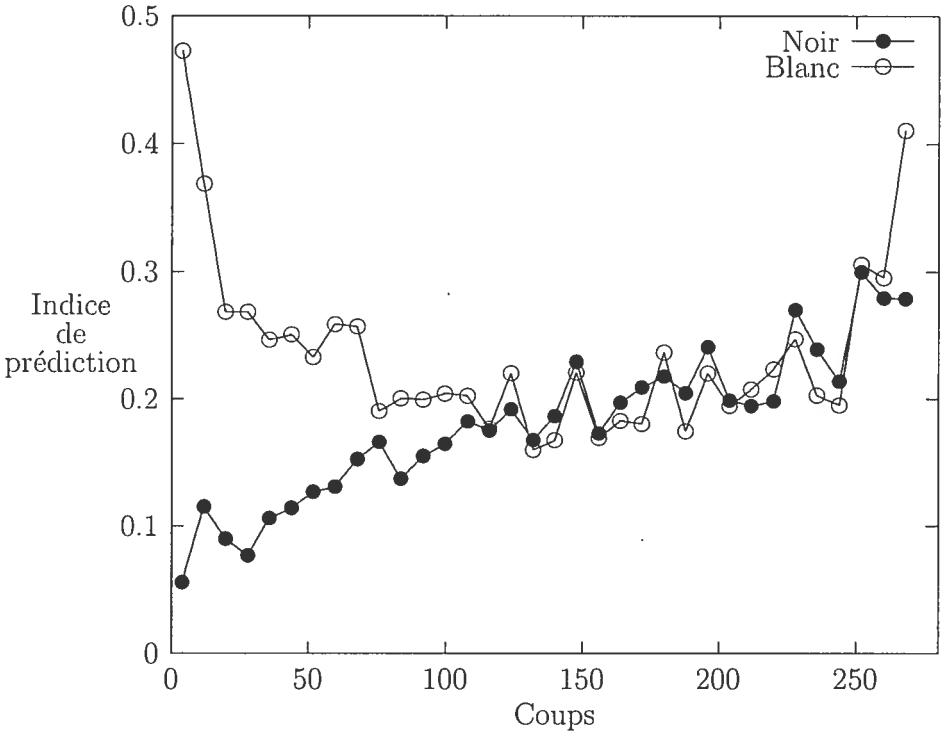

FIG. 5.1 – L'indice de prédiction, sans Minimax.

un meilleur pouvoir prédictif avec notre fonction d'évaluation ?"

Avant de répondre à cette question, regardons les coûts en temps de calcul que représente une telle fouille. À partir d'une position donnée, il faut nécessairement évaluer toutes les positions résultantes de coups pour prendre une décision éclairée. Sans fouille, il nous faut donc, en moyenne, 200 évaluations (en estimant 200 coups possibles, en moyenne, par position).

La façon la plus simple de faire la fouille consisterait à évaluer les coups, et pour chacun des n le plus hautement évalués, évaluer à nouveaux tous les coups et continuer ainsi pour k niveaux. La figure 5.2 montre que le nombre d'évaluations nécessaires croît très rapidement (de l'ordre de $200n^k$).

Puisque nous voulons faire mieux que sans fouille, k doit être au moins 2. De même, il faut que le coups joué soit fréquemment dans les n plus haut coups si on veut voir une amélioration. Considérant les résultats de la section précédente,

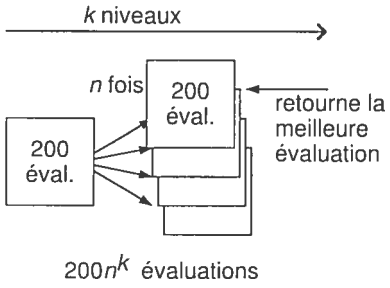

FIG. 5.2 – Le nombre d'évaluation, Minimax simple.

il nous faut un n d'au moins 20. On voit rapidement que cette technique est prohibitive quand aux ressources nécessaires.

La théorie du go nous dit que le bon coup pour un joueur est souvent un bon coup pour l'autre joueur. Aussi, les améliorations a Minimax telles que le "killer move" semblent indiquer que les bons coups non-joués resteront bons pour les coups suivants. Nous pouvons donc faire une seule fois l'évaluation de tous les coups possibles, puis faire une fouille en ne considérant que les mêmes n coups les plus haut évalués. La figure 5.3 illustre cette façon de faire. On obtient un nombre d'évaluations de l'ordre de $200 + n^k$. Ainsi, il nous est possible de faire l'évaluation de notre fonction d'évaluation avec $n = 20$ et $k = 2$.

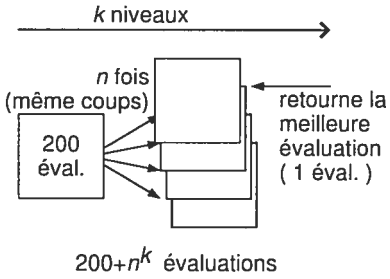

FIG. 5.3 – Le nombre d'évaluation, Minimax avec mêmes évaluations.

Malheureusement, on n'obtient pas les résultats escomptés. Avec $n = 20$ et $k = 2$, en utilisant l'évaluation initiale pour faire l'ordonnancement des coups à chaque niveau, on obtient un pouvoir prédictif moyen de 0.214. Ceci constitue une baisse de performance par rapport au cas sans fouille Minimax. La figure 5.4 montre la comparaison entre les deux résultats.

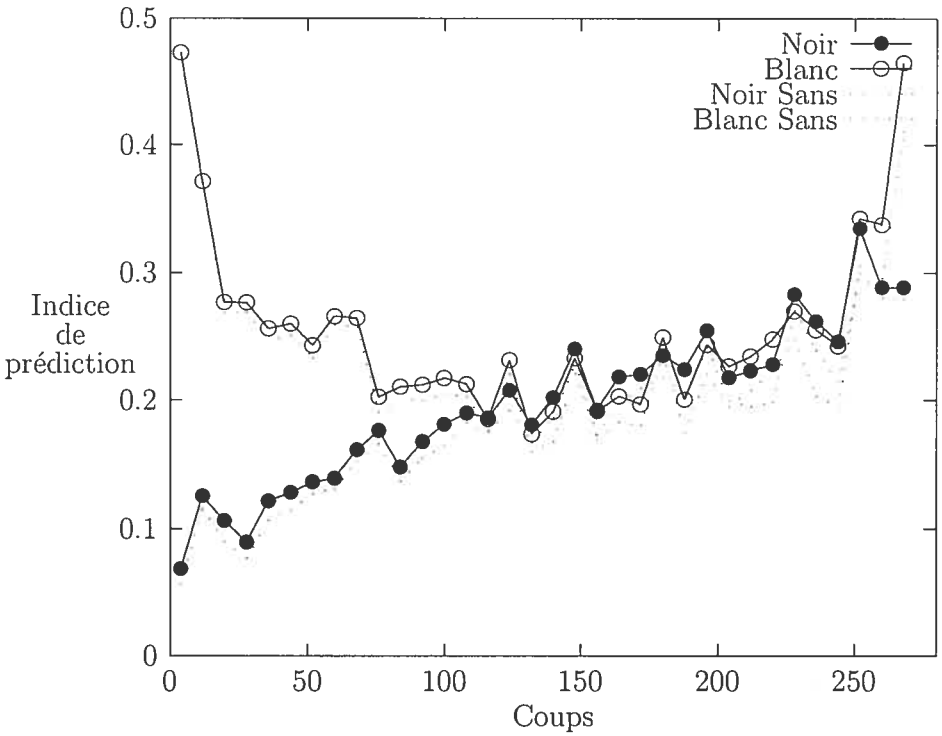

FIG. 5.4 – L'indice de prédiction, avec Minimax.

On voit rapidement qu'avec ces paramètres, une fouille Minimax ne semble d'aucune utilité. Plusieurs raisons peuvent expliquer ce résultat. Tout d'abord, il est possible que notre fonction d'évaluation ne soit efficace qu'à comparer des positions différant d'une seule pierre. Plus on fait une fouille profonde, plus les valeurs évaluées correspondent à des positions variées. Il est possible que la fonction d'évaluation soit moins efficace à comparer celles-ci. Ensuite, le fait d'avoir d'abord évalué les positions après un seul coup puis d'utiliser cet ordonnancement pour choisir les coups dans la fouille pourrait être une source d'erreur. L'utilisation

appropriée de la fouille Minimax pour améliorer le pouvoir prédictif demanderait des ressources prohibitives.

5.2.3 La distance au coup précédent

Notre fonction d'évaluation donne une valeur au goban sans considérer le coup précédent et est utilisée pour prévoir le coup suivant. Pour effectuer cette tâche, si on utilise l'information sur le coup précédent, on peut obtenir beaucoup plus facilement un pouvoir prédictif supérieur. La technique est simple : il suffit de considérer que le coup joué est très souvent à proximité du coup précédent. L'algorithme pour ordonner les coups devient donc : pour tout les coups, calculer la distance de Manhattan entre le coup précédent et le coup considéré. Ordonner selon la distance.

Avec une telle technique, on obtient le résultat suivants : pouvoir prédictif moyen de 0.125, soit clairement mieux que notre technique. Cependant, il ne faut pas oublier que cette technique utilise plus d'information pour obtenir ce résultat. La figure 5.5 montre le pouvoir prédictif de la distance selon la couleur et le nombre de coups.

5.2.4 Petit ensemble d'entrées et distance au coup précédent

Pour maximiser le pouvoir prédictif, on peut combiner ces deux approches. La façon utilisée consiste à ordonner les coups en deux listes, selon la distance et selon l'évaluation, puis à donner à chaque coup une valeur égale à la somme de leur classement dans chaque liste. Les coups obtenant la plus petite valeur sont préférés. Avec cette combinaison, on obtient, tel qu'attendu, un résultat meilleur que celui obtenu avec chacune des approches indépendantes.

Le pouvoir prédictif moyen des deux techniques combinées est de 0.102. Ce résultat montre clairement que l'on peut rejeter une grande partie des coups possible en se trompant très peu souvent. La figure 5.6 illustre le pouvoir prédictif selon la couleur et le nombre de coups.

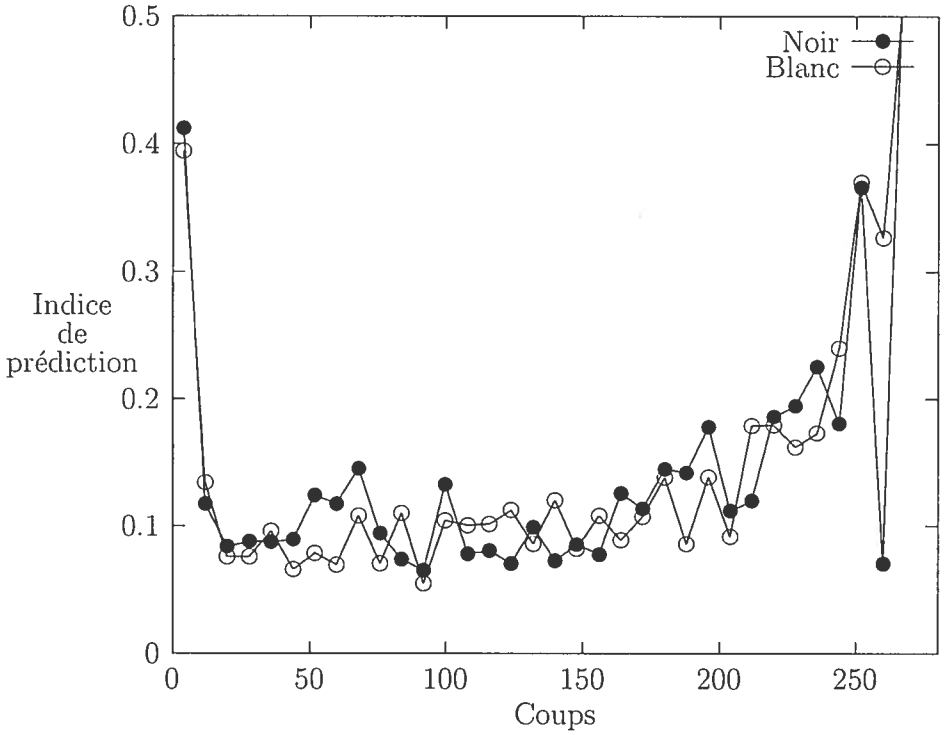

FIG. 5.5 – L'indice de prédiction, distance au coup précédent.

5.3 Les yeux et l'algorithme de Benson

Le deuxième ensemble de poids appris est obtenu avec un réseau à deux étages, et un ensemble élargi d'entrées, c'est-à-dire : toutes les entrées pertinentes de la section précédente, plus l'information sur les yeux. Le bloc de connection utilisé a 3 niveaux ayant respectivement 16, 12 et 8 neurones. Le nombre d'entrées du bloc, 16, et de sorties du bloc, 8, découle encore une fois directement du nombres de caractéristiques.

Plus en détail, 4 items influenceront les valeurs des entrées supplémentaires :

1. la vie inconditionnelle d'une chaîne
2. le nombre de régions saines pour chaque chaîne
3. le nombre de régions vides saines pour chaque chaîne
4. le nombre de petites régions noire(blanche)-adjacentes à chaque chaîne noires(blanches)

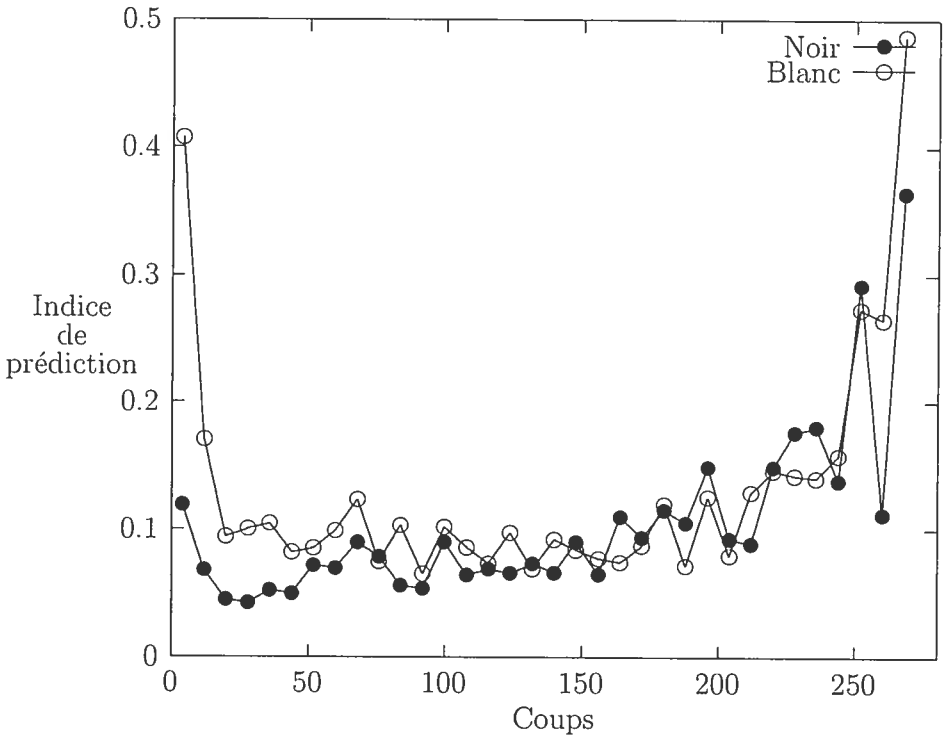

FIG. 5.6 – L'indice de prédiction, petit ensemble d'entrées et distance.

Ces éléments sont ceux décrits par Benson [Ben76] dans "Life in the Game of Go". Nous les illustrerons avec la figure 5.7.

L'ensemble de a,b et d forme une region noire-adjacente (small black-enclosed region, selon Benson), c'est à dire un ensemble de points connexes où chaque point est soit blanc, soit vide et adjacent à une pierre noire. Nous écrirons aussi x-adjacente, lorsque l'on parle de la couleur x, sans la nommer. Dans ce cas, nous utiliserons la notation $\overline{x}$ pour dénoter la couleur opposée.

La façon donc notre programme calcule les caractéristiques des chaînes se fait toujours pour une région d'une seule couleur. Or une région x-adjacente contient des pierres $\overline{x}$ et des points vides. Nous avons alors introduit le concept de région vide x-adjacente. a et b sont ainsi des régions vides noir-adjacentes. Une région x-adjacente est donc l'ensemble de regions vides x-adjacentes et de chaînes $\overline{x}$.

L'avantages de cette approche est double. On peut calculer l'état de nos régions

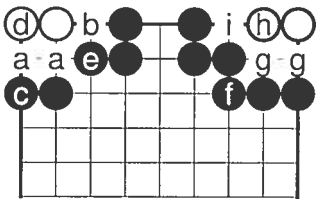

FIG. 5.7 – L'algorithme de Benson.

et chaînes facilement, de façon itérative. De plus, une chaîne peut avoir une région vide x-adjacente sans avoir de région x-adjacente. Dans ce cas, le réseau à une information partielle valide. Il sait que sur le goban se trouvent certains éléments pouvant conduire à une région x-adjacente.

Ce que Benson appelle "healthy regions", nous appellerons *région saines*. En se référant à la figure 5.7 on dit de la région formée par g, h et i qu'elle est saine pour la chaîne f, car tous les points libres de g, h et i sont des libertés de f.

De la même façon, nous dirons d'une région vide x-adjacente quelle est saine pour une chaîne si tous ces points sont adjacents à la chaîne. Ainsi, a est saine pour c, et b est saine pour e. Cependant, la région vide x-adjacente formée de a,b et d n'est saine ni pour c, ni pour e.

L'idée est que pour évaluer une position non-terminale et détecter la vie conditionnelle plutôt qu'inconditionnelle, il est utile de donner, au réseau de neurones, les éléments ayant servi au calcul de la vie inconditionnelle.

Nous avons modifié les 5 entrées utilisées précédemment de façon à réduire ce nombre à 4. Plutôt que d'avoir une entrée 1 ou -1 indiquant la couleur de la chaîne et quatre autres entrées entre 0 et 1, les autres entrées auront un signe positif ou négatif selon la couleur de la chaîne. Aussi, nous ajoutons 4 entrées supplémentaires pour l'algorithme de Benson. Ces entrées supplémentaires, pour une chaîne, y, de couleur x sont donc :

- Le nombre de régions x-adjacentes saines touchant y.
- Le nombre de régions vides x-adjacentes saines touchant y.

- Le nombre de régions x-adjacentes touchant y.
- Le nombre de régions $\overline{x}$-adjacentes touchant y.

Ces quatres valeurs ont aussi été transformées pour être entre 0 et 1. La fonction utilisée est la suivante : $f(x) = (1 - (\frac{1}{2x}))$ où x est le nombre de régions et $f(0) = 0$. De plus, si l'algorithme de Benson nous dit que la chaîne est inconditionnellement vivante, nous modifions les entrées pour le nombre de régions x-adjacentes saines et le nombre de régions vides x-adjacentes saines à 1. Ce qui correspond à la même chose qu'un très grand nombre de régions saines.

5.3.1 Résultats avec l'*algorithme de Benson*

Malgré le grand potentiel que semblait avoir l'information sur les yeux, cette information n'amène aucune amélioration des résultats. Dans le meilleur cas, les résultats atteignent tout juste la qualité de ceux obtenus sans cette information. De plus, l'apprentissage est beaucoup plus lent, car l'espace des paramètres est plus grand. Les éléments suivants expliquent cette absence de résultats :

- Insuffisance de données

  Pour que notre fonction d'évaluation ait un meilleur pouvoir prédictif, il faudrait que l'information sur les yeux soit souvent utilisée. Or, si on regarde l'ensemble de toutes les parties, une très faible proportion des chaînes sont des petites régions x-adjacentes. Or, sans petite région x-adjacente, aucune région n'est saine et aucun œil n'est détecté.

- Petit horizon d'apprentissage

  Lors de l'apprentissage, uniquement les positions résultant d'un seul coup, à partir de la position de départ, sont considérées. Or, le fait qu'un œil puisse être créé plus tard est souvent la raison justifiant le choix du coup.

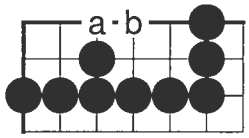

FIG. 5.8 – La détection des yeux.

Si on regarde la figure 5.8, il est relativement aisé de voir que le joueur noir fera certainement deux yeux. Aucun joueur blanc compétent n'ira se risquer à jouer dans ce coin. Afin de simplifier le texte, appellons le joueur noir, *Noir*, et le joueur blanc, *Blanc*. Pour n'importe quel coup de Blanc, Noir peut jouer en a ou b et être assuré de faire deux yeux. Blanc ne peut même pas en tirer une bonne menace de ko. Pourtant, cette région n'est même pas une petite région x-adjacente. L'algorithme de Benson dit, avec raison, qu'en plusieurs coups (15) Blanc pourrait capturer la chaîne noire. Pour que notre réseau détecte la sécurité de cette chaîne, une fouille serait nécessaire. Pire encore, une fouille au moment du jeu ne serait pas suffisante. Si on veut que le réseau apprenne à quel moment il est bien de jouer près d'un œil, il faudrait aussi faire la fouille lors de l'apprentissage.

Plusieurs autres exemples pourraient être montrés, mais la base reste : au go, les coups menant à la vie inconditionnelle ou à la mort d'une chaîne ne sont pas joués, car les deux joueurs connaissent l'issue de la bataille et s'abstiennent.

Ainsi, l'information sur les yeux selon l'algorithme de Benson est rarement présente, et lorsqu'elle l'est, elle est inutile car l'apprentissage a été insuffisant. Si on voulait faire l'apprentissage automatique d'une fonction d'évaluation pour le go qui tienne compte de cette information, il serait nécessaire de faire une fouille à partir de chaque position de l'ensemble de données. De plus, la fonction d'évaluation n'aurait une performance supérieure que si on l'utilise aussi en conjonction avec une fouille lors de l'évaluation.

CHAPITRE 6

L'UTILITÉ DU PROJET DANS UN CADRE PLUS LARGE

Dans ce chapitre, nous discuterons des différentes utilisations possibles d'une fonction d'évaluation et dans quel cadre plus large elle devrait s'insérer. Nous commencerons par présenter les éléments nécessaires pour construire un adversaire solide, puis nous verrons comment notre fonction peut s'y intégrer.

6.1 La complexité du go

L'algorithme de base pour un joueur électronique, pour tout jeu à deux joueurs, sans hasard, est le même. On fait une fouille dans l'arborescence des positions pour trouver une séquence qui garantit la victoire. Nous appelons cette recherche la fouille Minimax.

6.1.1 La fouille Minimax

L'algorithme pour la fouille Minimax de base est le suivant : nous assignerons à différentes positions des valeurs numériques selon les positions auxquelles elles peuvent mener. Une position terminale (où la partie est terminée) gagnante pour Noir vaudra 1. Une position terminale gagnante pour Blanc vaudra -1. Une position terminale nulle vaudra 0.

Pour une position non-terminale, au tour de Noir, on assigne la valeur de la position résultante possible la plus élevée. Ainsi, sous un jeu parfait, Noir choisira de jouer le coup qui lui donne le meilleur résultat (gain si possible, sinon nul, et autrement défaite).

De la même façon, pour une position non-terminale, Blanc à jouer, on assigne la valeur de la position résultante possible la moins élevée. Ainsi, sous un jeu parfait, Blanc choisira de jouer le coup qui donne le moins bon résultat possible à Noir.

Ainsi, il devient possible d'évaluer n'importe quelle position de façon récursive, en évaluant la valeur des positions résultantes, pour chaque coup. La figure 6.1 illustre cet arbre de fouille, pour un goban 1×3

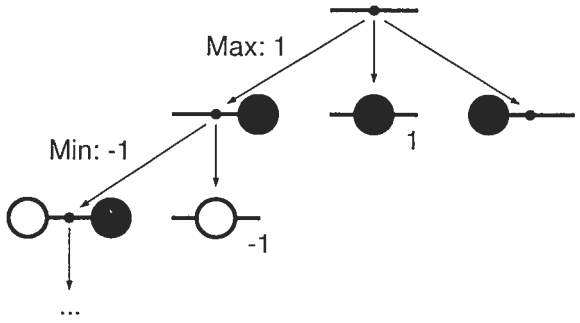

FIG. 6.1 – La fouille Minimax.

À partir du goban vide, un des coups (au centre) mène à une position terminale gagnante pour Noir. Il est donc clair que la valeur de la position est 1, car Noir jouera certainement ce coup, pour gagner. Lorsqu'une branche de l'arbre mène ainsi à une valeur maximale, la fouille est simplifiée. En général, il faut cependant explorer plusieurs branches.

Le nombre de positions à analyser croît comme n^k, où n est le nombre moyen de positions légales à partir d'une position donnée, et k la profondeur sur laquelle il faut faire la fouille. Pour un goban de 19×19, si on prend des valeurs approximatives de 200 coups possibles en moyenne, et une partie durant environ 200 coups, on peut estimer le nombre de positions à évaluer, pour faire une fouille complète. Avec, $k = 200$ et $n = 200$, il faudra analyser environ 200^{200} parties. Une telle fouille dépasse les possibilités actuelles des machines les plus puissantes, et selon toute probabilité, les dépassera encore longtemps.

Le portrait brossé jusqu'ici de la complexité du go semble bien sombre. Cependant, en considérant uniquement la taille de l'espace de recherche, nous oublions encore des points très importants. Les sous-sections suivantes montrent divers éléments qu'il faut aussi considérer.

6.1.2 La détection des positions terminales

Même si la puissance de calcul nécessaire pour faire une fouille complète telle que décrite précédemment était disponible, nous n'aurions toujours pas un adversaire électronique parfait. En effet, il nous faudrait aussi avoir une façon de détecter quand une partie est terminée.

Pour terminer la partie, les deux joueurs passent leur tour. La motivation pour passer son tour est de savoir qu'aucun point supplémentaire ne peut être gagné par chacun des deux joueurs. Dans bien des parties jouées par des humains, il reste encore quelques points qui pourraient être gagnés par un des deux joueurs. Cependant, il arrive qu'aucun des deux joueurs ne voit cette possibilité, et qu'elle ne soit pas jouée.

Pour faire un joueur électronique, il nous faut donc nécessairement écrire un module spécialisé qui peut détecter quand une position est terminale. Une technique, peu efficace, consisterait à faire une fouille encore plus longue pendant laquelle diverses attaques sont tentées, et où les joueurs emplissent leur propres territoire, jusqu'à ce qu'ils doivent passer.

6.1.3 Les fouilles locales

Jusqu'ici, nous avons discuté d'éléments qui rendent le go plus complexe, pour les humains comme pour les machines. Cependant, plusieurs caractéristiques du go le rendent aisé pour l'humain tout en rendant la programmation d'un algorithme difficile. Le premier élément que nous verrons est le découplage entre différentes régions du goban. Si on observe la figure 6.2, on peut se demander si les pierres blanches marquées ▲ ainsi que les noires marquées △ peuvent vivre et former du territoire.

Pour répondre à cette question, une fouille est possible. Nous avons vu le coût important qu'une fouille représente en temps de calcul. Cependant, on peut observer que toutes les pierre non-marquées sont inconditionnellement vivantes. Ainsi, le goban est séparé en deux régions indépendantes. Toute séquence de coups qu'un

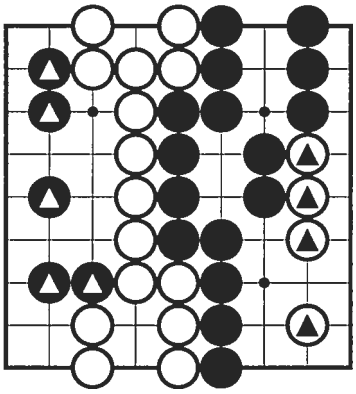

FIG. 6.2 – Le découplage entre diverses régions.

joueur peut trouver pour faire du territoire avec les pierres marquées peut être trouvé en ne considérant que les coups d'un même coté du goban. Ainsi, si la fouille demandait n^k positions, elle pourrait maintenant prendre de l'ordre de $(\frac{n}{2})^{\frac{k}{2}}$.

Il existe une exception importante à cette indépendance des régions. Si un ko survient d'un des deux cotés du goban, une menace, faite de l'autre coté, influence l'issue de la bataille locale. Ainsi, les ko couplent les deux régions indépendantes. Néanmoins, en identifiant les possibilités de ko, et en comptant le nombre de menaces de ko de chaque coté, il est possible d'utiliser ce découplage pour réduire grandement la complexité de la fouille. En effet, l'ordre dans lequel les menaces sont faites de chaque coté influence peu le résultat.

Il est donc possible de réduire grandement le calcul nécessaire pour les fouilles. Même lorsque les régions ne sont pas complètement indépendantes, de faire une fouille locale peut indiquer le bon ordre des coups dans une région donnée.

6.1.4 La vie et la mort

Une autre habileté importante pour un joueur de go est de savoir identifier l'*état* d'une chaîne. L'état d'une chaîne peut être *vivante*, si elle vivra nécessairement ; *morte*, si elle sera nécessairement capturée ; ou dépendre d'un ko. La figure 6.3

illustre l'importance de cette habileté.

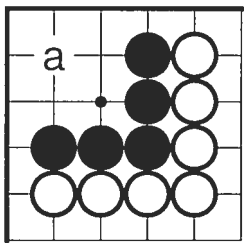

FIG. 6.3 – La vie et la mort.

Si la chaîne blanche de la figure 6.3 n'a pas d'œil ailleurs sur le goban dont le coin est montré, elle vivra seulement si elle peut capturer le coin. À l'inverse, si Noir réussi à protéger le coin, il est en bonne position. La seule question devient alors : le coin peut-il vivre, et comment ? Dans ce cas-ci, si Noir joue en a d'abord, le coin est inconditionnellement vivant. Au contraire, si Blanc joue en premier à cet endroit, l'issue dépendra d'un ko. Il est intéressant de noter que la plupart des joueurs de go connaissent cet état sans pour autant savoir défendre le coin contre toute attaque. Ainsi, ils peuvent jouer le bon coup, et espérer que l'adversaire ne testera pas leur habileté à défendre le coin.

Il est donc inutile de faire une fouille complète incluant tout le goban. Une fouille locale dans le coin permettra d'identifier cet état. Par la suite, on peut se demander, ou calculer, si une bataille pour un ko est une option ou non. De plus, plusieurs techniques permettent d'accélérer les cas où l'on tente de déterminer la vie et la mort, particulièrement les courses pour capturer, ou *semeai*. Un semeai est illustré à la figure 6.4

Chacune des deux chaînes marquées d'un triangle risque d'être capturée. Noir tente de jouer a, b, c, et d, avant que Blanc ne puisse jouer 1, 2, 3, et 4, et vice-versa. Le premier joueur à jouer peut donc s'assurer de capturer la chaîne ennemie. Le point intéressant, ici, est que l'ordre des coups en a, b, c, et d ainsi qu'en 1, 2, 3,

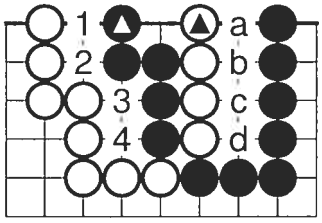

FIG. 6.4 – Une course pour capturer.

et 4 n'a pas d'importance. Comme il y a 4! permutations possibles pour chaque couleur, une fouille qui essaie chaque combinaison prendra $4!^2$, ou 576 fois plus de temps. Ainsi, un algorithme de fouille optimal doit pouvoir identifier un semeai et faire uniquement la partie pertinente de la fouille.

6.1.5 La connexité des chaînes

Parfois, ce qu'il est facile (pour un humain) et utile de voir, qu'une fouille normale ne considère pas, est la *connexité* des chaînes. Par connexité on entend que deux ou plusieurs chaînes peuvent logiquement être considérées comme une seule, dans la mesure où le joueur pourra nécessairement les connecter, si le besoin se présente.

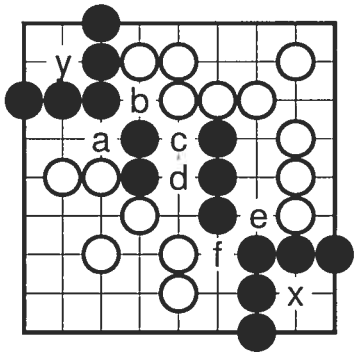

FIG. 6.5 – Un exemple de connexité.

Sur la figure 6.5, on peut voir que Noir peut faire un œil en x et un en y, mais pas deux en aucun des deux endroits. Pour vivre, il faut donc que Noir connecte ces deux chaînes. Une fouille qui tenterait de trouver une façon certaine de connecter pourrait être très longue et demander l'analyse de nombreuses positions.

Une façon simple d'aborder le problème est de voir que si blanc joue en a, Noir peut jouer en b, et vice-versa. De la même façon, si Blanc joue en c, Noir peut jouer en d et vice-versa. La même observation est valide pour e et f. Quoi que fasse Blanc, Noir pourra connecter ses chaînes ensemble et vivre. Encore une fois, cependant, l'exception est que Noir doit maintenant considérer que Blanc a 3 menaces de ko supplémentaires. C'est-à-dire que Blanc menace de couper Noir s'il peut jouer deux coups consécutifs où il veut.

Il reste cependant qu'il est plus simple d'analyser le goban en considérant les chaînes noires connectées et blanc ayant des menaces de ko supplémentaires qu'en considérant tous les ordres différents de coups en a, b, c, d, e, et f. De cette façon, on voit qu'il nous faudra aussi un module spécialisé qui identifie la connexité des chaînes, si on espère développer un adversaire électronique solide et rapide.

6.1.6 L'initiative

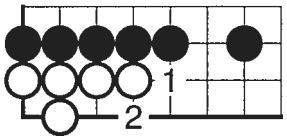

FIG. 6.6 – Un coup sente.

Deux concepts importants au go dont nous n'avons pas parlé jusqu'ici sont *sente* et *gote*. On dit d'un coup qu'il est sente s'il force l'adversaire à répondre localement à une menace, au risque de perdre beaucoup de points. La figure 6.6 illustre un coup sente. À partir de cette position, si Noir joue en 1, Blanc doit répondre immédiatement en 2, pour ne pas perdre un œil et voir son coin capturé.

En effet, si Noir peut jouer 1, puis 2, le coin devient un cadavre blanc.

L'avantage d'un coup sente est que le joueur qui le joue peut, s'il le désire, immédiatement obtenir les points de territoire associé et continuer à jouer d'autres coups sente. Ainsi, on appelle initiative le fait qu'un joueur puisse choisir où les prochains coups seront. Chaque joueur tente de garder l'initiative. À l'inverse, on dit d'un coup qu'il est gote s'il ne force pas l'adversaire à répondre localement, ou si la séquence résultante donne l'initiative à l'autre joueur. La figure 6.7 montre un coup gote.

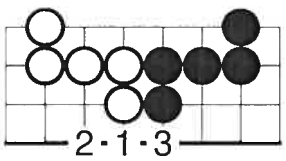

FIG. 6.7 – Un coup gote.

Après Noir 1, Blanc 2, la pierre noire en 1 se retrouve en atari et Noir doit la protéger en jouant 3. Dès lors, Blanc a l'initiative et peut jouer où bon lui semble. Ainsi, Noir en jouant son coup gote a fait un point, mais a perdu l'initiative.

Lors de la fin de partie, de pouvoir identifier la valeur de chaque coup ainsi que le type de coup, gote ou sente, permet de jouer un excellent go, sans avoir à considérer de multiples combinaisons de coups répartis sur tout le goban. Ainsi, nous voyons encore une technique nécessaire pour développer un bon, et efficace, adversaire électronique.

6.2 L'ordonnancement des coups dans les fouilles

Nous avons vu plusieurs éléments que devrait considérer un bon programme jouant au go. Nous verrons maintenant comment notre fonction peut améliorer la performance de ceux-ci.

Quelles que soient les améliorations ou optimisations apportées, tout programme

jouant au go aura une fouille globale, même si elle est très limitée. Lors de la fouille, il faut choisir un ordre dans lequel essayer les différents coups possibles. L'ordre optimal, bien sûr impossible, serait certainement d'essayer le bon coup en premier. Néanmoins, si le bon coup est essayé plus tôt, on réduit grandement le temps nécessaire pour la fouille grâce à la *coupe alpha-beta*.

La coupe alpha-beta permet d'accélérer la fouille en ne considérant pas les autres coups possibles pour un joueur, lorsqu'un coup est suffisamment bon pour que l'adversaire évite cette position. Nous illustrerons avec la position montrée à la figure 6.8.

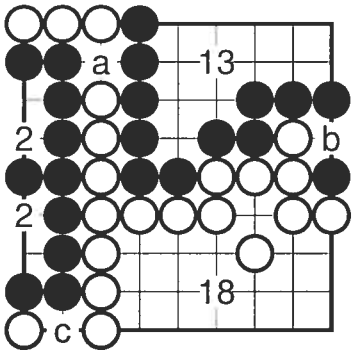

FIG. 6.8 – Une position illustrant la coupe alpha-beta.

À partir de cette position, les trois coups a,b,c sont intéressants. Le reste du territoire est décidé, et nous avons indiqué les points correspondants. La figure 6.9 montre l'arbre de fouille à partir de cette position.

Si Blanc joue en a, quelle que soit la réponse de Noir, il s'assure la victoire par un point. La valeur à la racine de l'arbre de fouille est donc, au plus, -1. En fouillant plus en avant, Blanc pourrait trouver un autre coup, qui lui donne un résultat meilleur, mais aucun coup menant à une valeur supérieure à -1 ne sera intéressant. Ainsi, en fouillant les positions, Blanc peut voir qu'après avoir joué b, Noir peut jouer a et obtenir un résultat de 3, bon pour Noir. Ainsi, tout coup choisi par Noir après le coup de Blanc en b vaudra au moins 3 points. Blanc n'a donc

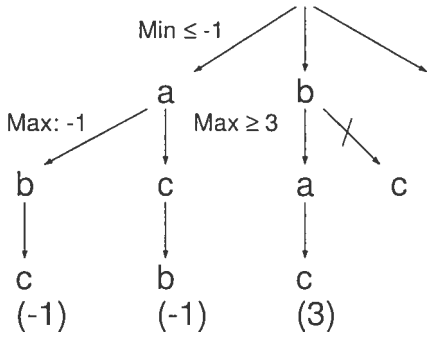

FIG. 6.9 – La coupe alpha-beta.

aucun intérêt à jouer b, quelles que soient les valeurs des autres positions possibles après b. Cela nous permet de ne pas faire la fouille des autres coups légaux après Blanc b.

Pour maximiser le nombre de positions coupées, il faut obtenir les valeurs les plus hautes pour Noir et les plus basses pour Blanc, le plus tôt possible. Il est donc important d'essayer les coups qui risquent le plus d'être les bons coups en premier. Notre fonction d'évaluation permet d'ordonner les coups de façon à ce que les bons coups arrivent en moyenne après dix pour-cent des coups possibles. Ainsi, nous augmentons grandement le nombre de coupes.

En plus de réduire globalement la complexité des fouilles globales, une bonne fonction d'évaluation permet d'améliorer la performance des modules pour

- La détection des positions terminales
- Les fouilles locales
- La vie et la mort

car celles-ci utilisent toutes des fouilles pour arriver à des résultats utiles. Ainsi, notre fonction d'évaluation est d'une utilité générale, en réduisant le temps de calcul des fouilles et, incidemment, en permettant des fouilles plus élargies. Elle pourra s'intégrer dans de nombreux modules constituant un adversaire électronique pour le go. En soit, elle ne donne pas une information directement utile. Elle permet plutôt d'accélérer les différentes fouilles nécessaires.

CONCLUSION

Après avoir rapidement présenté le jeu de go et les réseaux de neurones, nous avons vu une technique utilisant les réseaux de neurones pour évaluer la valeur d'une position au go. Cette technique à l'avantage d'apprendre automatiquement à partir d'un banque de parties jouées par des humains, où seuls les coups joués sont disponibles. L'élément nouveau de cette approche est de construire un réseau dont la topologie dépend de la position sur le goban. De plus, nous avons opté pour donner au réseau de l'information de très haut niveau, ce qui a bien fonctionné. À l'inverse, nous avons vu que de l'information trop spécialisée n'était pas nécessairement utile.

Nous avons ensuite montré les résultats obtenus. Bien que ces résultats soient intéressants et démontrent un apprentissage possible, ils ne sont pas fracassants. C'est en combinant notre fonction d'évaluation avec une technique commune, jouer à proximité du coup précédent, que l'on obtient un algorithme relativement efficace pour prévoir les bons coups. Une technique qui semblait plus prometteuse, utilisant de l'information sur les yeux, n'a pas donné les résultats escomptés.

Le développement de programmes passablement complexes a été nécessaire pour permettre que l'apprentissage soit efficace et prenne un temps raisonnable. Il faut se rappeler que l'information de plus haut niveau est plus difficile à identifier et que de conserver l'état du goban à jour, de façon itérative, n'est pas une tâche facile. De même, tout programme roulant sur seize ordinateurs pour plusieurs semaines demande une bonne infrastructure, et une certaine surveillance.

Les résultats obtenus nous ont permis de conclure que

- l'apprentissage automatique est possible à partir d'une banque de coups joués,
- un réseau de neurones dont la topologie varie avec la position peut être utilisé,
- toute l'information sur les chaînes n'est pas pertinente,
- il est plus facile de prédire le coup suivant en utilisant la position des coups précédents qu'en essayant de simplement donner une valeur à la position,

- bien qu'une fouille doive pouvoir aider la performance d'un évaluateur, la puissance de calcul nécessaire peut être prohibitive.

Par la suite, nous avons montré comment cet algorithme pourrait servir dans le cadre plus large du développement d'un adversaire électronique complet. Les fouilles globales et locales, la vie et la mort, ainsi que la détection des positions terminales peuvent toutes bénéficier d'une accélération, en utilisant notre fonction d'évaluation.

Clairement, à elle seule, notre approche ne suffit pas à jouer au go, même au plus faible niveau. On sait que beaucoup de travail reste à faire avant que les ordinateurs ne puissent jouer un go de haut calibre. Néanmoins, l'approche présentée pourrait servir à des programmes relativement puissants ou inspirer de nouvelles techniques. Nous l'espérons certainement.

BIBLIOGRAPHIE

[Ass91a] The American Go Association. The complete aga rules of go, September 1991. http ://www.usgo.org/resources/downloads/completerules.pdf.

[Ass91b] The American Go Association. The concise aga rules of go, April 1991. http ://www.usgo.org/resources/downloads/conciserules.pdf.

[Ben76] David B. Benson. Life in the game of go. *Information Sciences*, 10 :17–29, 1976. http ://www.cs.ualberta.ca/ games/go/seminar/notes /020717/benson.pdf.

[Dah99] Fredrik A. Dahl. Honte, a go-playing program using neural nets, 1999. http ://www.moyogo.com/downloads/dahl99honte.pdf.

[Enz96] Markus Enzenberger. The integration of a priori knowledge into a go playing neural network, September 1996. http ://www.cgl.ucsf.edu/go/Programs/neurogo-html/NeuroGo.html.

[Fot93] David Fotland. Knowledge representation in the many faces of go, 1993. ftp ://ftp-igs.joyjoy.net/Go/computer/mfg.tex.Z.

[LS80] David Lichtenstein and Michael Sipser. Go is polynomial-space hard. *Journal of the ACM*, 27(2) :393–401, April 1980.

[Mül02] Martin Müller. Computer go. *Artificial Intelligence*, 134 :145–179, 2002.

[NNSS94] Peter Dayan Terrence Nicol N. Schraudolph and J. Sejnowski. Temporal difference learning of position evaluation in the game of go. *Neural Information Processing Systems*, 6 :817–824, 1994. http ://www.gatsby.ucl.ac.uk/ dayan/papers/sds94.html.

[PDC94] Patrick Corr Paul Donnelly and Danny Crookes. Evolving go playing strategy in neural networks, 1994. ftp ://ftp-igs.joyjoy.net/Go/computer/egpsnn.ps.Z.

[RCAD02] Julian Churchill Richard Cant and David Al-Dabass. A hybrid artificial intelligence approach with application to games. *IEEE02*, 2002.

http ://ducati.doc.ntu.ac.uk/uksim/dad/webpagepapers/IEEE02/ CantChrchill -Hawaii.pdf.

[Rob83] J. M. Robson. The complexity of go. *Information Processing, proceedings of IFIP Congress*, pages 413–417, 1983.